ELSEVIER

Science of Computer Programming 44 (2002) 253–292

Science of Computer Programming

www.elsevier.com/locate/scico

A hierarchy of communication models for Message Sequence Charts

A.G. Engels, S. Mauw, M.A. Reniers*

Department of Mathematics and Computer Science, Technische Universiteit Eindhoven (TU/e), P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

Received 28 August 2000; received in revised form 21 November 2001; accepted 22 November 2001

Abstract

In a Message Sequence Chart (MSC) the dynamical behaviour of a number of cooperating entities is depicted. An MSC defines a partial order on the communication events between these entities. This order determines the physical architecture needed for implementing the specified behaviour, such as a FIFO buffer between each of the entities. In a systematic way, we define 50 communication models for MSC and we define what it means for an MSC to be implementable by such a model. Some of these models turn out to be equivalent, in the sense that they implement the same class of MSCs. After analysing the notion of implementability, only ten classes remain, for which we develop a hierarchy. We also develop algorithms to check whether a given MSC belongs to such a class. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Message Sequence Charts; Semantics; Implementation; Validation; Buffering; Communication models; Hierarchy

1. Introduction

In recent years much attention has been paid to graphical languages for the visualisation of communication traces in distributed systems. One of the most popular classes of formalisms for this purpose is the class of sequence charts. Of those, Message Sequence Chart (MSC) [27] has been standardised by the International Telecommunication Union (ITU) as Recommendation Z.120 [15]. Two important reasons for the popularity of MSCs are that they provide a clear intuition to both engineers and designers and at the same time posses a well-defined semantics.

* Corresponding author.
E-mail address: m.a.reniers@tue.nl (M.A. Reniers).

PII: S0167-6423(02)00022-9

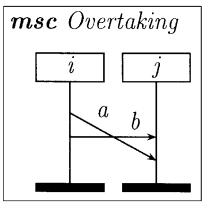

Fig. 1. MSC with overtaking messages.

An example of an MSC is in Fig. 1, showing entities (also called instances) i and j which exchange messages a and b.

The meaning of an MSC is often expressed in terms of execution traces. According to the semantics, the first event to occur is the sending of message a, followed by the sending and reception of message b, after which, message a is received. This simple example already shows that in an MSC there is no a priori assumption about the type of buffering (if at all) taking place between the communicating entities. Clearly, this MSC can not be implemented by a system where communication takes place using one single FIFO buffer. Furthermore, one may notice that this MSC describes a system where communication is essentially asynchronous, since the sending and reception of message a must be two different events.

The assumptions about the buffering of messages in MSC, are in contrast with the situation in a specification language such as SDL [13], where every entity has its own FIFO input buffer. Since MSC and SDL are often used in conjunction, there is a need to clarify this seemingly contradictory situation.

When considering restricted communication mechanisms, it is very natural to identify subclasses of MSC which exactly satisfy such buffering properties. One can consider the class of FIFO *buffered* MSCs, the class of *synchronous* MSCs, etc. In fact, the Interworkings language [22,24] is the latter class.

When considering Interworkings simply as a subset of MSC, an obvious question to ask is: what exactly is the distinction between synchronous and asynchronous MSCs? Or, phrased a little bit differently, how can we formally characterise the class of synchronous MSCs? Finding an answer to this question is not too difficult. An MSC is synchronous if and only if in every execution trace of the MSC there are no events between every pair of corresponding send and receive events.

But, how about the question whether an MSC can be implemented using only one FIFO buffer. And what, if we are allowed to use a number of FIFO buffers? This gives rise to a more general question. Can a given MSC be implemented by means of a given communication model? This is the question which will be studied in this paper.

Thereto, we define the notion of *communication model*, we present a formal semantics of MSC based on partial orders, and we define criteria for an MSC being implementable in a given communication model. We will not study the complete range of all possible communication models, but we will single out a number of interesting options, which we systematically derive by looking at the *locality* of the buffers

between the communicating entities. One can, e.g., assume one single FIFO buffer for the complete system, or a FIFO buffer between each pair of entities, etc. We will also take into account the difference between output buffers and input buffers, since in practice this distinction is often made.

Apart from studying the fundamental concepts behind the implementability of MSCs, there are also more practical motivations for the research presented here. First of all, the formal relation between scenario specifications in MSC and complete system specifications in a Formal Description Technique, such as SDL [13], is an important issue in the software engineering process. Not only the derivation of MSC scenarios from a Formal Description Technique, but also the synthesis of a complete specification from a collection of MSC scenario specifications is considered of great importance by many authors and tool builders (see [1,8,10,17,18,20,21,26,28,29]). This naturally leads to the question which MSCs can and which MSCs cannot be implemented in the given specification language.

One can also study the same question from a different perspective, namely, given an arbitrary MSC, how can we restrict (or extend) its semantics in such a way that it can be implemented in a given communication model. This question is partly studied by Alur et al. [2], who also derived supporting tools. Our starting point, however, will be that we consider the standard MSC semantics.

This brings us to the variety of ways in which MSCs are used, some of which are essentially different. We mention the distinction between *hot* and *cold* MSCs (see [7]) where (parts of) MSCs must or may occur in the implementation and we mention the difference between positive and negative use of MSC (an MSC must occur or is not allowed to occur). Finally, some users apply MSC to specify one single trace, while others consider the complete set of traces generated by an MSC.

Because an MSC has (in general) a number of traces, there are in fact two questions we can ask: Can an MSC be implemented in a certain communication model at all, that is, is there *some* trace of the MSC that is implementable (the *weak case*), and can an MSC be implemented in a certain communication model without losing behaviour, that is, is *each* trace of the MSC implementable (the *strong case*). We will discuss both questions in this paper.

Since all implementation relations introduced in this paper identify subclasses of the class of MSCs, it is interesting to know how these classes relate. The answer to this question is formulated as a hierarchy of communication models for MSCs. We will also present characterisations, which can be used to create algorithms to check to which class a given MSC belongs. An overview of the results, with both the hierarchy and the characterisations can be found in Fig. 23.

We present our research in the following way. In Section 2.1 we introduce a subset of the MSC language called Basic Message Sequence Charts, and give a formal semantics based on partial orders. The communication models which we study are defined in Section 2.2. In order to be able to deal with two distinct buffers between two communicating entities, we will extend the standard partial order semantics in Section 2.3. The definition of implementability of a single trace with respect to a communication model is given in Section 2.4. In Section 3, we classify traces according to their implementability. This work is lifted to the level of MSCs in Section 4, where we first

study the strong case (Section 4.1), and then the weak case (Section 4.2). The overall picture combining the strong and the weak case is given in Section 4.3. We also give a number of characterisations of the implementability relations, which make it possible to determine the implementability of a given MSC algorithmically (see Section 5). Section 6 contains a comparison with related literature and in Section 7 we summarise our findings and discuss options for further research.

2. Message Sequence Charts

2.1. Basic Message Sequence Charts

The MSCs studied here consist of a collection of instances (or entities) with a number of messages attached to them. These are known as Basic Message Sequence Charts, but in this paper we use the term MSCs to denote them.

An example of an MSC can be seen in Fig. 2. It consists of vertical lines, denoting the various communicating entities and arrows between these instances, denoting exchanged messages.

We allow messages from an instance to itself, but we only consider closed systems, that is, we do not consider messages to the environment. We assume that the names of the instances and messages are unique. Therefore, the instances to which a message is attached are determined uniquely by the message name.

In this section we explain the semantical foundations of MSC. We use a partial order on the events of an MSC to express the semantics. In literature several ways to define the semantics of MSC are proposed [9,11,16,19,23]. The process algebra approach [25] has been standardised as Annex B to ITU recommendation Z.120 [14]. The partial order representation [2] used in this paper coincides with most of these proposals for the class of Basic Message Sequence Charts. We also define the traces expressed by an MSC.

The easiest way to express the semantics of such a simple MSC is by using a partial order on the events that are comprised in an MSC. Depending on the particular dialect of the MSC language, one can assign different classes of events to an MSC. For

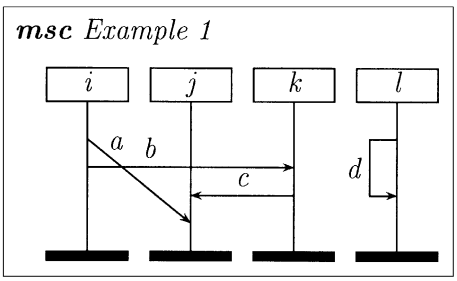

Fig. 2. Example MSC.

example, in Interworkings [22,24] every message is considered to be a single event. There is no buffering, and thus communication is synchronous.

In MSC [15], messages are divided into two events, the output and the input of the message. The output of message m is denoted by $!m$ and the input by $?m$. The only assumption about the implementation of communication is that an output precedes its corresponding input. An MSC describes a partial order on output and input events.

Definition 1 (MSC). An MSC is a quintuple $\langle I, M, \mathsf{from}, \mathsf{to}, \{<_i\}_{i\in I}\rangle$, where I is a finite set of instances, M is a finite set of messages, from and to are functions from M to I, and $\{<_i\}_{i\in I}$ is a family of orders. For each $i\in I$ it is required that $<_i$ is a total order on $\{!m \mid \mathsf{from}(m)=i\}\cup\{?m \mid \mathsf{to}(m)=i\}$. We use the shorthand $E_{\mathsf{msc}}(M)$ to denote the set $\{!m, ?m \mid m\in M\}$.

In the above definition, $\mathsf{from}(m)$ denotes the instance which sends message m. Likewise, $\mathsf{to}(m)$ denotes the instance which receives message m. Given an instance i, the order $<_i$ denotes in which order the events attached to instance i occur.

MSC *Example 1* from Fig. 2 is thus represented by the quintuple

$$\langle I, M, \mathsf{from}, \mathsf{to}, \{<_i\}_{i\in I}\rangle,$$

where

- $I=\{i,j,k,l\}$,
- $M=\{a,b,c,d\}$,
- $\mathsf{from}(a)=\mathsf{from}(b)=i$, $\mathsf{from}(c)=k$, and $\mathsf{from}(d)=l$,
- $\mathsf{to}(a)=\mathsf{to}(c)=j$, $\mathsf{to}(b)=k$, and $\mathsf{to}(d)=l$, and
- $<_i=\{(!a,!b)\}$, $<_j=\emptyset$, $<_k=\{(?b,!c)\}$, and $<_l=\{(!d,?d)\}$.

The partial order denoting the semantics of an MSC k is derived from two requirements. First, the order of the events per instance is respected, and second, a message can only be received after it has been sent. The first requirement is formalised by defining the *instancewise* partial order $<_k^{\mathsf{inst}}$:

$$<_k^{\mathsf{inst}} = \bigcup_{i\in I} <_i,$$

and the second requirement is formalised by the *output-before-input* order $<_k^{\mathsf{oi}}$:

$$<_k^{\mathsf{oi}} = \{(!m, ?m) \mid m \in M\}.$$

Now, we define the partial order induced by the MSC as the transitive closure (denoted by $^+$) of the instancewise order and the output-before-input order. For an MSC k, we denote this order by $<_k^{\mathsf{msc}}$ or by $<^{\mathsf{msc}}$ if k is known from the context.

Definition 2. For a given MSC $k=\langle I, M, \mathsf{from}, \mathsf{to}, \{<_i\}_{i\in I}\rangle$, the relation $<_k^{\mathsf{msc}}$ is defined by $<_k^{\mathsf{msc}} = (<_k^{\mathsf{inst}} \cup <_k^{\mathsf{oi}})^+$.

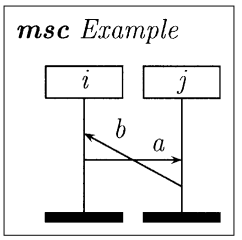

Fig. 3. Inconsistent MSC.

For MSC *Example 1* we thus have $<^{\mathsf{oi}} = \{(!a,?a),(!b,?b),(!c,?c),(!d,?d)\}$ and therefore $<^{\mathsf{msc}} = \{(!a,!b),(!a,?a),(!b,?b),(?b,!c),(!c,?c),(?c,?a),(!d,?d),(!a,?b),(!a,!c),(!a,?c),(!b,!c),(!b,?c),(?b,?c),(!c,?a)\}$.

It may be the case that $<^{\mathsf{msc}}$ does not define a partial order, due to cyclic dependencies of the events. Such an MSC is said to contain a *deadlock*, or is called *inconsistent*. An example of an inconsistent MSC is given in Fig. 3. In Z.120 [15], inconsistent MSCs are considered illegal, and in [4] an algorithm is described for determining whether a given MSC is consistent. In the remainder of this paper we consider consistent MSCs only, which implies that $<^{\mathsf{msc}}$ is a partial order.

Definition 3 (*Traces*). A trace t over a set of events E is a total order on these events.

We denote the ith element of a trace t by t_i, and its length by $|t|$. As a consequence of the above definition we can associate with each trace t an order $<_t^{\mathsf{trace}}$. This order is useful in expressing that a certain trace t is actually a trace of an MSC k.

Definition 4 (*Trace order*). For a trace t over a set of events E we define an order $<_t^{\mathsf{trace}}$ on E, for all $1 \leqslant i \leqslant |t|$ and $1 \leqslant j \leqslant |t|$, by $t_i <_t^{\mathsf{trace}} t_j \Leftrightarrow i < j$. Thus, an event e_i is smaller, according to $<_t^{\mathsf{trace}}$, than an event e_j if and only if it occurs earlier in the trace.

Definition 5 (*msc-trace*). A trace t is said to be an msc-trace of the MSC k if and only if it is defined over the set of events $E_{\mathsf{msc}}(M)$ of k, and $<_k^{\mathsf{msc}} \subseteq <_t^{\mathsf{trace}}$.

Lemma 6. *For an MSC k over M, and events $e, e' \in E_{\mathsf{msc}}(M)$, we have $e <_t^{\mathsf{trace}} e'$ for all msc-traces t of k if and only if $e <_k^{\mathsf{msc}} e'$.*

Proof. The 'if'-part is trivial. For the 'only if'-part we use contraposition. Suppose that $e \not<_k^{\mathsf{msc}} e'$. Then the relation $<_k^{\mathsf{msc}} \cup \{(e',e)\}$ does not contain a cycle. Thus, it can be extended to a total order $<$. Because $<^{\mathsf{msc}} \subseteq <$, $<$ will be the order $<_t^{\mathsf{trace}}$ of some msc-trace t of k. In this msc-trace we will have $e' <_t^{\mathsf{trace}} e$, and thus $e \not<_t^{\mathsf{trace}} e'$. □

2.2. Models for communication

In this section we discuss possible architectures for realising an MSC. We consider only communication models consisting of FIFO buffers for the output and input of messages. For msc-traces, we define what it means to be implementable on some architecture.

The particular communication models which we are interested in are constructed of entities that communicate with each other via FIFO buffers. We assume that the buffers have an unbounded capacity. We discern two uses of buffers, namely for the output and for the input of messages.

A second distinction can be made based on the locality of the buffer. From most global to most local we distinguish the following types:

- global: A global FIFO buffer: All messages from all instances pass this buffer.
- inst: A FIFO buffer, local to an instance: All messages sent (or received) by one single instance go through the same buffer.
- pair: A FIFO buffer, local to two instances: All messages that are sent from one specific instance to another specific instance go through this buffer.
- msg: A FIFO buffer, local to a message: There is one buffer for every message.

This last model, a buffer per message, is a specific architecture to catch up the cases in which the buffers do not behave like FIFO queues, but as random-access buffers. Taking into account the assumption that messages are unique, it can easily be seen that it is equivalent to a global random-access buffer. A communication model with only a random-access buffer represents the implied model of the MSC standard: the only assumption made about the implementation of communication is that output precedes input, no more, and no less.

Finally, we consider the following possibility:

- nobuf: There are no buffers; communication is synchronous.

We assume that all output buffers are of the same type, and similarly that all input buffers are of the same type. This results in four possibilities for the output as well as for the input. Adding the possibility of using no buffer at all, we have a total of 25 possible architectures, as shown in Fig. 4. To denote the elements of this scheme, we use the notation (X,Y), where X denotes the type of output buffer, and Y the type of input buffer.

In Fig. 5 we give examples of a physical architecture of three communication models. A circle denotes an instance, an open rectangle denotes an output buffer, a filled rectangle denotes an input buffer, and an arrow denotes a communication channel. Each example contains three instances. The first example illustrates the (nobuf, global) model. There is no output buffer, and one universal input buffer. As there is no output buffer, the messages go straight into the input buffer. This single buffer could be regarded as an output buffer as well, so this example is an illustration of (global, nobuf) too if we replace the input buffer by an output buffer. The second example shows the (global, inst) model. There is one general output buffer and every instance has a local input buffer. The third architecture is an example of the (pair, pair) model.

output \ input	nobuf	global	inst	pair	msg
nobuf	●	●	●	●	●
global	●	●	●	●	●
inst	●	●	●	●	●
pair	●	●	●	●	●
msg	●	●	●	●	●

Fig. 4. Communication models.

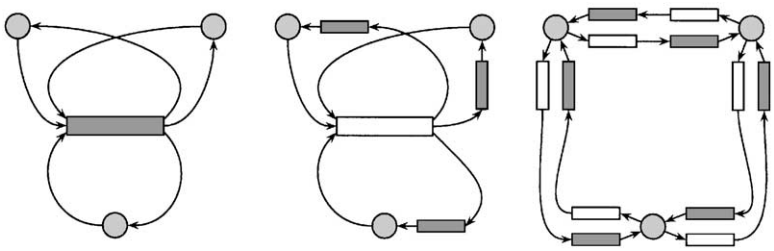

Fig. 5. Some models: (nobuf, global), (global, inst) and (pair, pair).

Please note that not all models described in Fig. 4 make equally sense. For example, the model (global, inst) (i.e., a shared medium for transmitting messages and an input buffer for each entity) is more natural than the exotic (global, pair) model.

Many of these architectures occur in practice as either the underlying communication architecture of a programming language or as a physical architecture. We give some examples of languages. The model (nobuf, nobuf) is typical for process algebraic formalisms based on synchronous communication, such as LOTOS [12] and ACP [5]. The specification language SDL [13,3], which is closely related to MSC, has as a general communication model (pair, msg), but if we leave out the *save* construct we obtain (pair, inst) and if we also do not consider the possibility of delayed channels, we have (nobuf, inst). Some examples of physical architectures are: an asynchronous complete mesh has a (nobuf, pair) architecture, and an Ethernet connection with locally buffered input and output behaves like (inst, inst).

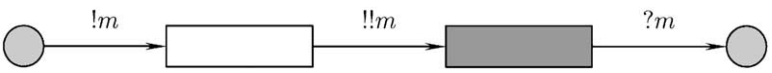

Fig. 6. Events associated with a communication.

2.3. Extending the semantics

In the previous section, we have seen that we consider communication models of communication in MSCs where each message passes at most two FIFO buffers. In order to reason about such communication models, we will extend the semantics of MSC in this section. In this extension of the semantics, a single communication of message m will be modeled by three events. These are the events $!m$, $!!m$, and $?m$. The intuition here is, as expressed in Fig. 6, that $!m$ denotes the putting of a message into an output buffer, $!!m$ is the transmission of the message from the output buffer to the appropriate input buffer, and $?m$ is the removal of the message from the input buffer. We assume these events to be instantaneous.

The intermediate transmit events $!!m$ play a crucial role in our description of the communication models. However, we have formulated the semantics of an MSC without using transmit events. In the remainder of this section, we will define a semantics of MSC in which the transmit event occurs. The approach is similar to the previously defined semantics.

The order $<_i$ is lifted in the trivial way to the set $E_{\mathsf{impl}}(M) = \{!m, ?m, !!m \mid m \in M\}$.

We define the *output-before-transmit-before-input* order by

$$<_k^{\mathsf{oti}} = \{(!m, !!m), (!!m, ?m) \mid m \in M\}$$

and the relation $<_k^{\mathsf{impl}}$ by adding the instancewise order on the MSC.

Definition 7. For a given MSC $k = \langle I, M, \mathsf{from}, \mathsf{to}, \{<_i\}_{i \in I}\rangle$, the order $<_k^{\mathsf{impl}}$ is defined by $<_k^{\mathsf{impl}} = (<_k^{\mathsf{inst}} \cup <_k^{\mathsf{oti}})^+$.

It is easy to see that $<^{\mathsf{msc}}$ is the restriction of $<^{\mathsf{impl}}$ to output and input events. From an operational point of view, one can say that an MSC describes a set of traces. We distinguish msc-traces and impl-traces: where an msc-trace denotes the ordering of output and input events ($!m$ and $?m$), an impl-trace denotes those of transmit events ($!!m$) as well.

Definition 8 (*impl-traces*). An *impl-trace* is the same as an msc-trace (see Definition 5), except for the fact that it contains transmit events as well.

Definition 9 (*impl-trace*). A trace t is said to be an impl-trace of the MSC k if and only if it is defined over the set of events $E_{\mathsf{impl}}(M)$ of k, and $<_k^{\mathsf{impl}} \subseteq <_t^{\mathsf{trace}}$.

An impl-trace can be turned into an msc-trace by removing all transmit events ($!!m$). If, for an impl-trace t this results in an msc-trace t', then t is said to be an extension

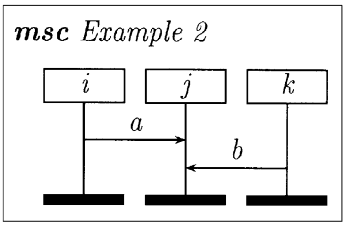

Fig. 7. Example MSC.

of t'. It is not hard to see that an impl-trace t is a trace of an MSC k if and only if the trace of which it is an extension is an msc-trace of the MSC and additionally the output-before-transmit-before-input order is respected: $<_k^{\mathsf{oti}} \subseteq <_t^{\mathsf{trace}}$.

The MSC *Example 2* from Fig. 7 implies the following orderings: $!a <^{\mathsf{msc}} ?a$, $!b <^{\mathsf{msc}} ?b$, and $?a <^{\mathsf{msc}} ?b$. The first two are implied by the $<^{\mathsf{oi}}$-order, the third by the $<^{\mathsf{inst}}$-order. The MSC has exactly three msc-traces: $!a?a!b?b$, $!a!b?a?b$, and $!b!a?a?b$. These msc-traces can be extended, by adding transmit events, to ten impl-traces, such as $!a!!a?a!b!!b?b$ and $!a!b!!b!!a?a?b$.

2.4. Implementability

The main question of this paper is, whether a system with a given implementation model can exhibit the behaviour described by a certain MSC. To answer this question, we first give a formal definition of what it means for a trace to have a certain implementability property. The definitions below can be seen as a formalisation of the notions introduced in Section 2.2.

Definition 10 (*Output-implementability*).

- nobuf-output: Every output event is directly followed by the corresponding transmit event. Thus, output and transmit events may be combined into one new event. An impl-trace t is nobuf-output implementable if and only if

$$\forall_{m\in M} \neg \exists_{e\in E_{\mathsf{impl}}(M)} !m <_t^{\mathsf{trace}} e <_t^{\mathsf{trace}} !!m.$$

- global-output: The order of two output events is respected by the corresponding transmit events. An impl-trace t is global-output implementable if and only if

$$\forall_{m,m'\in M} !m <_t^{\mathsf{trace}} !m' \Rightarrow !!m <_t^{\mathsf{trace}} !!m'.$$

- inst-output: The order of any two output events from the same instance is respected by the corresponding transmit events. An impl-trace t is inst-output implementable if and only if

$$\forall_{m,m'\in M} \mathsf{from}(m) = \mathsf{from}(m') \Rightarrow (!m <_t^{\mathsf{trace}} !m' \Rightarrow !!m <_t^{\mathsf{trace}} !!m').$$

- pair-output: The order of two output events with the same source and the same destination, is respected by the corresponding transmit events. An impl-trace t is pair-output implementable if and only if

$$\forall_{m,m' \in M} \; \mathsf{from}(m) = \mathsf{from}(m') \wedge \mathsf{to}(m) = \mathsf{to}(m') \Rightarrow (!m <_t^{\mathsf{trace}} !m' \Rightarrow !!m <_t^{\mathsf{trace}} !!m').$$

- msg-output: An impl-trace t is always msg-output implementable.

For msg-output implementability we can remark that it can be put in line with the three definitions preceding it, by restating it as

$$\forall_{m,m' \in M} \; m = m' \Rightarrow (!m <_t^{\mathsf{trace}} !m' \Rightarrow !!m <_t^{\mathsf{trace}} !!m').$$

For nobuf-output implementability such a translation is not possible; this is qualitatively another definition. Also note that, because $<^{\mathsf{trace}}$ is a total order, $!m <_t^{\mathsf{trace}} !m' \Rightarrow !!m <_t^{\mathsf{trace}} !!m'$ is equivalent to both $!m <_t^{\mathsf{trace}} !m' \Leftrightarrow !!m <_t^{\mathsf{trace}} !!m'$ and $!m <_t^{\mathsf{trace}} !m' \Leftarrow !!m <_t^{\mathsf{trace}} !!m'$.

The input implementabilities are defined analogously.

Definition 11 (*Input-implementability*).

- nobuf-input: An impl-trace t is nobuf-input implementable if and only if

$$\forall_{m \in M} \neg \exists_{e \in E_{\mathsf{impl}(M)}} !!m <_t^{\mathsf{trace}} e <_t^{\mathsf{trace}} ?m.$$

- global-input: An impl-trace t is global-input implementable if and only if

$$\forall_{m,m' \in M} \; !!m <_t^{\mathsf{trace}} !!m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m'.$$

- inst-input: An impl-trace t is inst-input implementable if and only if

$$\forall_{m,m' \in M} \; \mathsf{to}(m) = \mathsf{to}(m') \Rightarrow (!!m <_t^{\mathsf{trace}} !!m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m').$$

- pair-input: An impl-trace t is pair-input implementable if and only if

$$\forall_{m,m' \in M} \; \mathsf{from}(m) = \mathsf{from}(m') \wedge \mathsf{to}(m) = \mathsf{to}(m') \Rightarrow (!!m <_t^{\mathsf{trace}} !!m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m').$$

- msg-input: An impl-trace t is always msg-input implementable.

Having defined formally the notions of output- and input-implementability, we now combine them and obtain our notion of communication model.

Definition 12. For $X, Y \in \{\mathsf{nobuf}, \mathsf{global}, \mathsf{inst}, \mathsf{pair}, \mathsf{msg}\}$, an impl-trace is said to be (X, Y)-*implementable* if and only if it is X-output implementable and Y-input

implementable. An msc-trace is said to be (X, Y)-*implementable* if and only if it can be extended (by adding $!!m$'s) to an impl-trace that is (X, Y)-implementable.

3. Classification of implementability of traces

To each of the communication models defined in the previous section we can associate the set of all traces that are implementable in the model. Based on the subset relation on these sets of traces, we can order communication models. We consider two models equivalent if they have the same set of implementable traces.

In Lemma 13 we give a classification of the notions of output-implementability. It states that a trace that is implementable on a certain architecture is also implementable on an architecture where these buffers are partitioned into buffers with a more restricted locality. For example, if a trace can be implemented on an architecture with one output buffer per instance, it can also be implemented on an architecture with an output buffer per pair of instances (provided the input buffers remain the same).

Lemma 13 (*Classification of output-implementability*).

- *Every* nobuf-*output implementable trace is* global-*output implementable*.
- *Every* global-*output implementable trace is* inst-*output implementable*.
- *Every* inst-*output implementable trace is* pair-*output implementable*.
- *Every* pair-*output implementable trace is* msg-*output implementable*.

Proof. For impl-traces this follows directly from the definitions. For msc-traces this follows from the definition plus the fact that it holds for impl-traces. □

Note that the counterpart of Lemma 13, where 'output' is replaced by 'input' also holds. The following lemmas give the orderings between the communication models.

Lemma 14.

- *Every* (inst, global)-*implementable msc-trace is* (inst, nobuf)-*implementable.*
- *Every* (global, global)-*implementable msc-trace is* (global, nobuf)-*implementable*.
- *Every* (pair, pair)-*implementable msc-trace is* (pair, nobuf)-*implementable*.
- *Every* (msg, msg)-*implementable msc-trace is* (msg, nobuf)-*implementable*.

Proof. We show the proof for (inst, global). The other proofs are roughly analogous. Let t be an msc-trace over the set of events $E_{\mathsf{msc}}(M)$, and let t' be an impl-trace that is an (inst, global)-implementable extension of t. It suffices to construct an (inst, nobuf)-implementable extension t'' of t. We create t'', for which we will prove that it is (inst, nobuf)-implementable, in the following way: Starting from t, for each message $m \in M$ we add the transmit event $!!m$ just before the input event $?m$. This t'' is nobuf-input implementable by definition, so it suffices to prove that t'' is inst-output implementable. Thereto, let $m, m' \in M$ such that $\mathsf{from}(m) = \mathsf{from}(m')$. We have to prove that $!m <^{\mathsf{trace}}_{t''} !m' \Rightarrow !!m <^{\mathsf{trace}}_{t''} !!m'$.

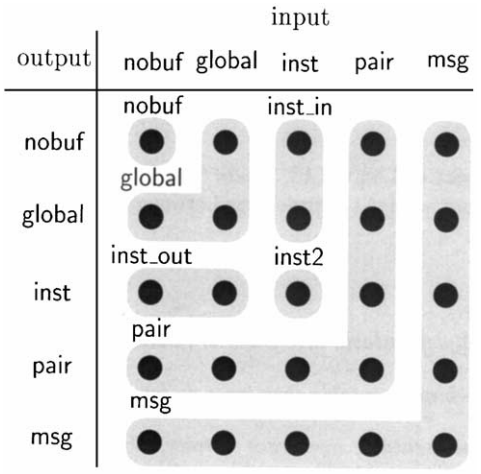

Fig. 8. Equivalence of communication models for traces.

Suppose that $!m <^{\mathsf{trace}}_{t''} !m'$. Then, since t'' is an extension of t, we have $!m <^{\mathsf{trace}}_{t} !m'$, and similarly, since t' is an extension of t, $!m <^{\mathsf{trace}}_{t'} !m'$. Using that t' is inst-output implementable and $\mathsf{from}(m) = \mathsf{from}(m')$ we have $!!m <^{\mathsf{trace}}_{t'} !!m'$. By using that t' is global-input implementable, we also have $?m <^{\mathsf{trace}}_{t'} ?m'$. Since t' is an extension of t we have $?m <^{\mathsf{trace}}_{t} ?m'$ and since t'' is an extension of t also $?m <^{\mathsf{trace}}_{t''} ?m'$. Since t'' is nobuf-input implementable, we obtain $!!m <^{\mathsf{trace}}_{t''} !!m'$, which completes the proof. □

Also Lemma 14 has a counterpart, which is obtained from Lemma 14 by switching the types of output and input buffers (i.e. by replacing (X, Y)-implementable with (Y, X)-implementable).

Next, we describe how the above lemmas are useful in ordering the models. Lemma 13 and its counterpart provide us with a partial order on the various implementations: Any (X, Y)-implementable trace is implementable by all communication models located to the right of or below (X, Y) in Fig. 4. Lemmas 13 and 14, together with their counterparts, give us the equivalences as expressed in Fig. 8 by means of the clustering of communication models.

For example, the models from the last column are equivalent. This can be seen as follows. Because of the analogue of Lemma 14, any (msg,msg)-implementable msc-trace is (nobuf, msg)-implementable, while Lemma 13 gives that any (nobuf, msg)-implementable msc-trace is (X, msg)-implementable, and every (X, msg)-implementable msc-trace is (msg, msg)-implementable.

Now we have reduced the number of communication models to only seven different classes. Of course, some of these could still be equivalent for other reasons than the above lemmas. That this is not the case, will be seen in Corollary 19 below. We name the equivalence classes as follows: nobuf, global, inst_out, inst_in, inst2, pair, msg (see Fig. 8).

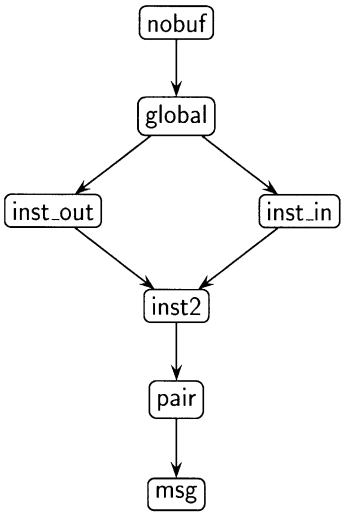

Fig. 9. Ordering of the communication models for traces.

Of these, the first two and last two will be clear immediately, inst_out means that there are instancewise output buffers and global or no input buffers, inst_in means that there are instancewise input buffers and global or no output buffers, and inst2 means that there are both an instancewise output buffer and an instancewise input buffer.

Theorem 15. *For msc-traces, the* 7 *communication models are ordered as shown in Fig.* 9.

Proof. This follows from the Lemmas 13 and 14 and their counterparts as explained above. □

Note that of these seven cases only inst2 is not of the form (X, nobuf) or (nobuf, X). As these forms imply that there is respectively no input buffer or no output buffer, of these seven cases only the case inst2 needs two buffers, all other cases can be modeled such that each message goes through at most one buffer.

It will prove useful to have a characterisation of these implementabilities (except for inst2 of course) that does not use *transmit* events.

Lemma 16. *Let* t *be an msc-trace over a set of events* $E_{\mathsf{msc}}(M)$. *Then*:

- t *is* nobuf-*implementable if and only if*

$$\forall_{m\in M} \neg \exists_{e\in E_{\mathsf{msc}}(M)} !m <_t^{\mathsf{trace}} e <_t^{\mathsf{trace}} ?m;$$

- t *is* global-*implementable if and only if*

$$\forall_{m,m'\in M} !m <_t^{\mathsf{trace}} !m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m';$$

- *t is* inst_out-*implementable if and only if*

$$\forall_{m,m' \in M} \mathsf{from}(m) = \mathsf{from}(m') \Rightarrow (!m <_t^{\mathsf{trace}} !m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m');$$

- *t is* inst_in-*implementable if and only if*

$$\forall_{m,m' \in M} \mathsf{to}(m) = \mathsf{to}(m') \Rightarrow (!m <_t^{\mathsf{trace}} !m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m');$$

- *t is* pair-*implementable if and only if*

$$\forall_{m,m' \in M} \mathsf{from}(m) = \mathsf{from}(m') \wedge \mathsf{to}(m) = \mathsf{to}(m') \Rightarrow (!m <_t^{\mathsf{trace}} !m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m');$$

- *t is always* msg-*implementable.*

Again note that because $<_t^{\mathsf{trace}}$ *is a total order,* $\forall_{m,m' \in M} !m <_t^{\mathsf{trace}} !m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m'$ *can be replaced by* $\forall_{m,m' \in M} !m <_t^{\mathsf{trace}} !m' \Leftrightarrow ?m <_t^{\mathsf{trace}} ?m'$ *without loss of correctness.*

Proof. The proofs for this are easily found by realising that an msc-trace is (X, nobuf)-implementable exactly if the conditions for X-output implementability hold with $!!m$ everywhere replaced by $?m$. □

4. Classification of MSCs

The use of MSCs in practice (and theory) is twofold. First, MSCs are often used to restrict the behaviour of communicating entities. In this use, it is the intention that the actual behaviour of the system is contained in the behaviour specified by the MSC. It does not mean that all behaviour of the MSC must be realised in the system. In this case only one of the msc-traces of the MSC has to be implementable in the given communication model. This notion of implementability is called *weak implementability*.

On the other hand, if the language MSC is used for the description of required behaviour (as for example in use cases), it is intended that each of the behaviours specified by the MSC is realised. In this case all msc-traces of the MSC have to be implementable in the given communication model. This notion of implementability is called *strong implementability*.

We first focus on strong implementability, then on weak implementability. After this we consider the relation between classes from the strong and weak spectrum.

4.1. Strong implementability

Definition 17. An MSC k is said to be *strongly X-implementable*, notation X_s-implementable, if and only if all msc-traces t of k are X-implementable.

From this definition it follows immediately that the ordering of the communication models for msc-traces as given in Fig. 9 also holds for MSCs as far as strong implementability is concerned (see Fig. 14). Next, we demonstrate that the communication

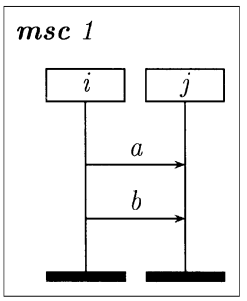

Fig. 10. MSC to distinguish nobuf_s- and global_s-implementability.

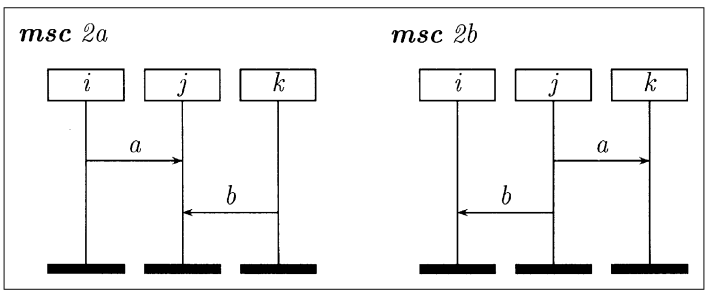

Fig. 11. MSCs to distinguish global_s-, $\mathsf{inst\_out}_s$-, $\mathsf{inst\_in}_s$-, and $\mathsf{inst2}_s$-implementability.

models, obtained by lifting them from the trace level to MSCs in the strong way, are indeed different. This is achieved by finding examples of MSCs that are in one class but not in another.

MSC 1 in Fig. 10 shows an example that is global_s-implementable, but not nobuf_s-implementable. It is not nobuf_s-implementable, because the msc-trace $!a\,!b\,?a\,?b$ is not. The input events necessarily have to be ordered in the same way as the output events, so it is global_s-implementable.

MSC 2*a* in Fig. 11 is $\mathsf{inst\_out}_s$-implementable, but not global_s-implementable due to the trace $!b\,!a\,?a\,?b$. That MSC 2*a* is $\mathsf{inst\_out}_s$-implementable can be seen as follows: All messages go through a different output buffer, so there is no problem with the output buffers at all. Similarly, MSC 2*b* (in the same figure) is $\mathsf{inst\_in}_s$-implementable, but not global_s-implementable due to the trace $!a\,!b\,?b\,?a$.

MSCs 2*a* and 2*b* show the difference between $\mathsf{inst\_out}_s$ and $\mathsf{inst\_in}_s$. MSC 2*a* is $\mathsf{inst\_out}_s$-implementable, as mentioned before, but not $\mathsf{inst\_in}_s$-implementable. The trace $!b\,!a\,?a\,?b$ is not inst_in-implementable, because the input events of instance *j* do not reach the input buffer in the order in which they are to be manipulated. For MSC 2*b* the reverse is the case: It is $\mathsf{inst\_in}_s$-implementable, but not $\mathsf{inst\_out}_s$-implementable. MSC 2*a* is $\mathsf{inst\_out}_s$-implementable and therefore also $\mathsf{inst2}_s$-implementable. We have already established that it is not $\mathsf{inst\_in}_s$-implementable. Similarly, MSC 2*b* is $\mathsf{inst\_in}_s$ and $\mathsf{inst2}_s$-implementable, but not $\mathsf{inst\_out}_s$-implementable. Together, these show that $\mathsf{inst\_out}_s$, $\mathsf{inst\_in}_s$ and $\mathsf{inst2}_s$ are all different.

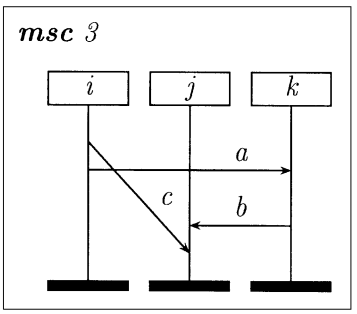

Fig. 12. MSC to distinguish $\mathsf{inst\_out}_s$- and pair_s-implementability.

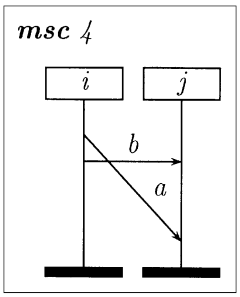

Fig. 13. MSC to distinguish pair_s- and msg_s-implementability.

One might suspect that the class of $\mathsf{inst2}_s$-implementable MSCs is simply equal to the intersection of the classes of $\mathsf{inst\_out}_s$-implementable and $\mathsf{inst\_in}_s$-implementable MSCs. This is not the case, as can easily be shown by combining the MSCs 2a and 2b into one MSC (see MSC 8 in Fig. 22).

MSC 3 in Fig. 12 is an example of an MSC that is pair_s-implementable, but not $\mathsf{inst2}_s$in-implementable. It is easy to see that it is pair_s-implementable, because each message goes through a different buffer. Its only msc-trace is $!c\,!a\,?a\,!b\,?b\,?c$. If we try to extend this to an inst2-implementable impl-trace t', we need to have $!!c <_t^{\mathsf{trace}} !!a <_t^{\mathsf{trace}} !!b <_t^{\mathsf{trace}} !!c$, which is impossible (the first $<_t^{\mathsf{trace}}$ is because of the inst-output implementability and $!c <_t^{\mathsf{trace}} !a$, the second is clearly true for every impl-trace of the MSC, and the third is because of the inst-input implementability together with $?b <_t^{\mathsf{trace}} ?c$).

Finally, MSC 4 in Fig. 13 shows the difference between pair_s- and msg_s-implementability. All other communication models are also pairwise different. This result is obtained due to the transitive closure of the ordering as presented in Fig. 14.

Together the examples used above show that if we look at strong implementability, the seven remaining implementation models are indeed different for MSCs, and thus that they are also different for msc-traces.

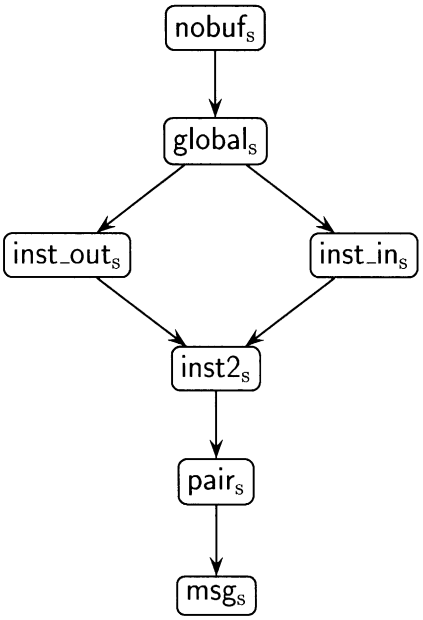

Fig. 14. Ordering scheme for strong implementability.

Theorem 18. *The communication models for strong implementability of Fig.* 14 *are different and these are ordered as expressed in Fig.* 14.

Proof. In the above text we have demonstrated by means of counterexamples that the communication models must be different. Also the ordering has been explained above. □

Corollary 19. *The classes* nobuf, global, inst_out, inst_in, inst2, pair, *and* msg *are different for traces.*

4.2. Weak implementability

Definition 20. An MSC k is said to be *weakly X-implementable*, notation X_w-implementable, if and only if there is an X-implementable msc-trace t of k.

As was the case for strong implementability, for weak implementability we also have the ordering as expressed in Fig. 9 as a starting point. However, using weak implementability, we do not have anymore that all communication models differ. To see this, we first give an alternative way to characterise some of the implementations and prove that these are equivalent to the original definition.

We will use some new relations (to denote these relations we will use the same type of symbols as we have used to denote partial orders) to give these characterisations.

The idea is that these new relations give a requirement that must be fulfilled by an msc-trace so as to be inst_out-implementable, inst_in-implementable or inst2-implementable. For example, to be inst_out-implementable, each time two messages m and m' come from the same instance, they must be received in the same order as the order in which they were sent. Because they are on the same instance, there will be some $<^{\mathsf{msc}}$-order between $!m$ and $!m'$. To ensure that the msc-trace has the receipts in the same order, we will have to add the equivalent order between $?m$ and $?m'$.

Definition 21. Let k be an MSC over the set of messages M. Then we define the relations $<_k^{\mathsf{io}}$ and $<_k^{\mathsf{ii}}$ on $E_{\mathsf{msc}}(M)$ and $<_k^{\mathsf{i2}}$ on $E_{\mathsf{impl}}(M)$ as follows:

$$
\begin{aligned}
<_k^{\mathsf{io}} &= (<_k^{\mathsf{msc}} \cup \{(?m, ?m') \mid m, m' \in M \wedge \mathsf{from}(m) = \mathsf{from}(m') \wedge !m <_k^{\mathsf{msc}} !m'\})^+,\\
<_k^{\mathsf{ii}} &= (<_k^{\mathsf{msc}} \cup \{(!m, !m') \mid m, m' \in M \wedge \mathsf{to}(m) = \mathsf{to}(m') \wedge ?m <_k^{\mathsf{msc}} ?m'\})^+,\\
<_k^{\mathsf{i2}} &= (<_k^{\mathsf{impl}} \cup \{(!!m, !!m') \mid m, m' \in M \wedge \mathsf{from}(m) = \mathsf{from}(m') \wedge !m <_k^{\mathsf{impl}} !m'\}\\
&\qquad \cup \{(!!m, !!m') \mid m, m' \in M \wedge \mathsf{to}(m) = \mathsf{to}(m') \wedge ?m <_k^{\mathsf{impl}} ?m'\})^+.
\end{aligned}
$$

A picture of these orders can be seen in Fig. 15. It shows an MSC together with its $<^{\mathsf{msc}}$, $<^{\mathsf{impl}}$, $<^{\mathsf{io}}$, $<^{\mathsf{ii}}$ and $<^{\mathsf{i2}}$ relations. For the last three, the orders that have been added when compared to $<^{\mathsf{msc}}$ or $<^{\mathsf{impl}}$ have been dashed, while the orders that caused these extra orders have been drawn fat.

The inst_out-implementable msc-traces of the MSC are also traces of the order $<_k^{\mathsf{oi}}$ as they respect the requirements for inst_out-implementability by definition, and vice versa. Basically this is what is expressed in Lemma 22.

Lemma 22. *Let t be an msc-trace of an MSC k. Then,*

- *t is* inst_out*-implementable if and only if $<_k^{\mathsf{io}} \subseteq <_t^{\mathsf{trace}}$;*
- *t is* inst_in*-implementable if and only if $<_k^{\mathsf{ii}} \subseteq <_t^{\mathsf{trace}}$;*
- *t is* inst2*-implementable if and only if there exists an extension t' of t such that $<_k^{\mathsf{i2}} \subseteq <_{t'}^{\mathsf{trace}}$.*

Proof. We only give the proof for the last proposition. The proofs for the first two propositions follow the same line.

First, suppose that t is inst2-implementable. Then we must prove that $<_k^{\mathsf{i2}} \subseteq <_{t'}^{\mathsf{trace}}$ for some impl-trace t' which is an extension of t. Let the impl-trace t' be an arbitrary inst2-implementable extension of t (the existence of such a trace follows trivially from Definition 12). Suppose that $e <_k^{\mathsf{i2}} e'$ for arbitrary events $e, e' \in E_{\mathsf{impl}}(M)$. Now it suffices to prove $e <_{t'}^{\mathsf{trace}} e'$. Since $e <_k^{\mathsf{i2}} e'$ we have the existence of events $e_1, \ldots, e_n$ such that $e \equiv e_1$, $e' \equiv e_n$ and for all $1 \leqslant i < n$ we have one of the following:

- $e_i <_k^{\mathsf{impl}} e_{i+1}$;
- $e_i \equiv !!m$ and $e_{i+1} \equiv !!m'$ for some $m, m' \in M$ such that $\mathsf{from}(m) = \mathsf{from}(m')$ and $!m <_k^{\mathsf{impl}} !m'$;
- $e_i \equiv !!m$ and $e_{i+1} \equiv !!m'$ for some $m, m' \in M$ such that $\mathsf{to}(m) = \mathsf{to}(m')$ and $?m <_k^{\mathsf{impl}} ?m'$.

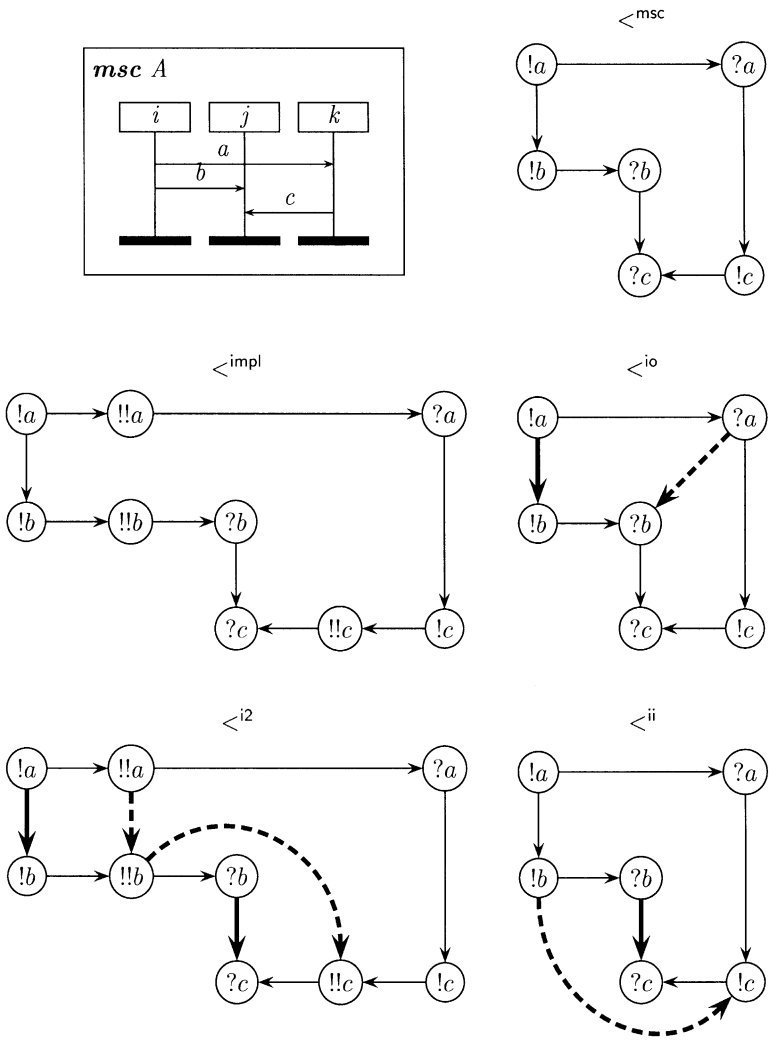

Fig. 15. Explanation of the $<^{\mathsf{io}}$, $<^{\mathsf{ii}}$ and $<^{\mathsf{i2}}$ relations.

In the first case we immediately have $e_i <_{t.}^{\mathsf{trace}} e_{i+1}$. Due to the fact that t' is an inst2-implementable impl-trace, and thus both inst-output and inst-input implementable, we can conclude that $e_i <_{t'}^{\mathsf{trace}} e_{i+1}$ for the second and third case as well (see Definitions 10 and 11). Since $<_{t'}^{\mathsf{trace}}$ is transitive we have $e <_{t'}^{\mathsf{trace}} e'$, which completes this part of the proof.

Second, suppose that $<_k^{\mathsf{i2}} \subseteq <_{t'}^{\mathsf{trace}}$ for some impl-trace t' which is an extension of t. We must prove that t is $(\mathsf{inst}, \mathsf{inst})$-implementable. Thereto, it suffices to show that t' is $(\mathsf{inst}, \mathsf{inst})$-implementable, i.e., that t' is inst-output implementable and inst-input implementable. We prove that t' is inst-output implementable, the proof that t' is inst-input implementable is analogous. Let $m, m' \in M$ such that $\mathsf{from}(m) = \mathsf{from}(m')$. Then

it suffices to show that $!m <^{\mathsf{trace}}_{t'} !m' \Rightarrow !!m <^{\mathsf{trace}}_{t'} !!m'$. Thus, suppose that $!m <^{\mathsf{trace}}_{t'} !m'$. Since $\mathsf{from}(m) = \mathsf{from}(m')$, we have $!m <^{\mathsf{msc}}_{k} !m'$. So $!!m <^{\mathsf{i2}}_{k} !!m'$. Because $<^{\mathsf{i2}}_{k} \subseteq <^{\mathsf{trace}}_{t'}$ we therefore have $!!m <^{\mathsf{trace}}_{t'} !!m'$. □

Thus far, we have seen that the order $<^{\mathsf{io}}_{k}$ has as its traces the inst_out-implementable msc-traces of MSC k. An MSC k is $\mathsf{inst\_out_w}$-implementable if and only if it has a trace t that is inst_out-implementable. Clearly, such a trace exists if and only if there is a trace for the order $<^{\mathsf{io}}_{k}$, in other words, if and only if $<^{\mathsf{io}}_{k}$ is cycle-free.

Theorem 23. *Let k be an MSC. Then,*

- *k is $\mathsf{inst\_out_w}$-implementable if and only if $<^{\mathsf{io}}_{k}$ is cycle-free;*
- *k is $\mathsf{inst\_in_w}$-implementable if and only if $<^{\mathsf{ii}}_{k}$ is cycle-free;*
- *k is $\mathsf{inst2_w}$-implementable if and only if $<^{\mathsf{i2}}_{k}$ is cycle-free.*

Proof. Follows immediately from Lemma 22. □

We use the alternative characterisations provided by Theorem 23 in the proof of the equivalence of the classes $\mathsf{inst\_out_w}$, $\mathsf{inst\_in_w}$, and $\mathsf{inst2_w}$.

Lemma 24. *Let k be an MSC over the set of messages M and let $m, m' \in M$. If $?m <^{\mathsf{io}}_{k} ?m'$, then $!!m <^{\mathsf{i2}}_{k} !!m'$*

Proof. Suppose that $?m <^{\mathsf{io}}_{k} ?m'$. Then by the definition of $<^{\mathsf{io}}_{k}$ we have the existence of events $e_1, \ldots, e_n$ such that $e_1 \equiv ?m$, $e_n \equiv ?m'$, and for $1 \leqslant i < n$ we have one of the following:

- $e_i <^{\mathsf{msc}}_{k} e_{i+1}$;
- $e_i \equiv ?p$, $e_{i+1} \equiv ?p'$ for some $p, p' \in M$ such that $\mathsf{from}(p) = \mathsf{from}(p')$ and $!p <^{\mathsf{msc}}_{k} !p'$.

In the second case we have $!!p <^{\mathsf{i2}}_{k} !!p'$ directly from Definition 21. In the first case we have a sequence of events where the smallest steps are due to $<^{\mathsf{inst}}$ or due to $<^{\mathsf{oi}}$. In this sequence any subsequence of events which are defined on the same instance can be replaced by one single step. As a result we have the existence of messages $m_1, \ldots, m_{n'}$ such that

$$e_i \leqslant^{inst} !m_1 <^{\mathsf{oi}} ?m_1 <^{\mathsf{inst}} !m_2 <^{\mathsf{oi}} ?m_2 <^{\mathsf{inst}} \cdots <^{\mathsf{inst}} !m_{n'} <^{\mathsf{oi}} ?m_{n'} \leqslant^{inst} e_{i+1},$$

where $f \leqslant^{inst} f'$ is short for $f <^{\mathsf{inst}} f'$ or $f \equiv f'$. Now we observe that we only have the following three possibilities for $<^{\mathsf{inst}}$:

- $!q <^{\mathsf{inst}} !q'$ for some $q, q' \in M$ such that $\mathsf{from}(q) = \mathsf{from}(q')$. Then also $!!q <^{\mathsf{i2}}_{k} !!q'$ by the definition of $<^{\mathsf{i2}}$.
- $?q <^{\mathsf{inst}} ?q'$ for some $q, q' \in M$ such that $\mathsf{to}(q) = \mathsf{to}(q')$. Then also $!!q <^{\mathsf{i2}}_{k} !!q'$, again by the definition of $<^{\mathsf{i2}}$.
- $?q <^{\mathsf{inst}} !q'$ for some $q, q' \in M$ such that $\mathsf{to}(q) = \mathsf{from}(q')$. Then $!!q <^{\mathsf{impl}}_{k} ?q <^{\mathsf{impl}}_{k} !q' <^{\mathsf{impl}}_{k} !!q'$, so clearly $!!q <^{\mathsf{impl}}_{k} !!q'$ and $!!q <^{\mathsf{i2}}_{k} !!q'$ (since $<^{\mathsf{i2}}_{k} \subseteq <^{\mathsf{impl}}_{k}$).

Thus, we obtain $!!e_i <^{\mathsf{i2}}_{k} !!e_{i+1}$ for all $1 \leqslant i < n$. Therefore $!!m <^{\mathsf{i2}}_{k} !!m'$. □

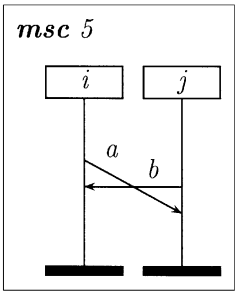

Fig. 16. MSC to distinguish nobuf_w- and global_w-implementability.

Lemma 25. *The communication models* $\mathsf{inst\_out}_w$, $\mathsf{inst\_in}_w$, *and* $\mathsf{inst2}_w$ *are equivalent.*

Proof. We show that each $\mathsf{inst2}_w$-implementable MSC is also $\mathsf{inst\_out}_w$-implementable. The reverse implication is trivial, and the proofs with $\mathsf{inst\_in}_w$ are analogous. From Lemma 23 we see that it suffices to prove that $<^{\mathsf{io}}$ is cycle-free if $<^{\mathsf{i2}}$ is cycle-free. We prove this using contraposition, so we assume that $<^{\mathsf{io}}$ has a cycle. Let $e_1 <^{\mathsf{io}} e_2 <^{\mathsf{io}} \ldots <^{\mathsf{io}} e_n <^{\mathsf{io}} e_1$ be an arbitrary cycle such that for every consecutive pair of events in the cycle, say $e_i <^{\mathsf{io}} e_{i+1}$, either $e_i <^{\mathsf{msc}} e_{i+1}$, and hence $e_i <^{\mathsf{i2}} e_{i+1}$, or $e_i \equiv ?m$, $e_{i+1} \equiv ?m'$ for some $m, m' \in M$ such that $!m <^{\mathsf{msc}} !m'$ and $\mathsf{from}(m) = \mathsf{from}(m')$ (any cycle can be extended to some cycle of this form by the addition of events).

If the first is always the case, then we have a cycle in $<^{\mathsf{msc}}$, so certainly in $<^{\mathsf{i2}}$. Now assume we have the second at least once in the cycle. In that case we have at least two input events in the cycle, say $?m$ and $?m'$. Then $?m <^{\mathsf{io}} ?m'$ and $?m' <^{\mathsf{io}} ?m$. Lemma 24 gives that this implies that $!!m <^{\mathsf{i2}} !!m'$ and $!!m' <^{\mathsf{i2}} !!m$, so $<^{\mathsf{i2}}$ has a cycle. □

Lemma 25 establishes that the classes $\mathsf{inst\_out}_w$, $\mathsf{inst\_in}_w$, and $\mathsf{inst2}_w$ are equivalent. In the remainder we denote this class by inst_w. The remaining models are all different. MSC 3 and MSC 4 in Figs. 12 and 13 show the difference between inst_w and pair_w, and pair_w and msg_w, respectively, in the weak case too (these MSCs have only one msc-trace, so their weak implementability equals their strong implementability). MSC 5 in Fig. 16 is global_w-implementable, but not nobuf_w-implementable. The msc-trace $!a\,!b\,?a\,?b$ is global-implementable, but because both output events must have been executed before any input event can be processed, there is no nobuf-implementable msc-trace.

MSC 6 in Fig. 17 is inst_w-implementable, but not global_w-implementable. It is not global_w-implementable, as can be seen thus: $!a <^{\mathsf{msc}} !b$, so if an msc-trace t of this msc is global-implementable, we must have $?a <_t^{\mathsf{trace}} ?b$. Because $!d <^{\mathsf{msc}} ?a$ and $?b <^{\mathsf{msc}} !c$, we get $!d <_t^{\mathsf{trace}} !c$. But we also have $?c <^{\mathsf{msc}} ?d$, and thus $?c <_t^{\mathsf{trace}} ?d$, from which it follows that t cannot be global-implementable. On the other hand, the msc-trace $!a\,!b\,!d\,?a\,?b\,!c\,?c\,?d$ is inst_out-implementable, so the MSC is inst_w-implementable.

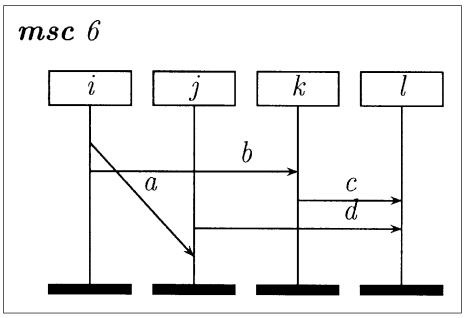

Fig. 17. MSC to distinguish global_w- and inst_w-implementability.

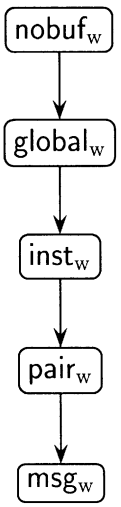

Fig. 18. Ordering scheme for weak implementability.

Theorem 26. *The communication models for weak implementability of Fig.* 18 *are all different and they are ordered as expressed in Fig.* 18.

Proof. The counterexamples that imply that the communication models are different are given above. The ordering of the models is inherited from the ordering of the communication models with respect to traces. Lemma 25 provides that the communication models $\mathsf{inst\_out}_w$, $\mathsf{inst\_in}_w$, and $\mathsf{inst2}_w$ are equivalent. □

4.3. Combining the strong and weak hierarchies

The relations between the classes in one of the two hierarchies have been studied extensively in the previous sections. We have 12 possible implementations left: nobuf_s, global_s, $\mathsf{inst\_out}_s$, $\mathsf{inst\_in}_s$, $\mathsf{inst2}_s$, pair_s and msg_s in the strong case, and nobuf_w, global_w, inst_w, pair_w and msg_w in the weak case. From the definitions of

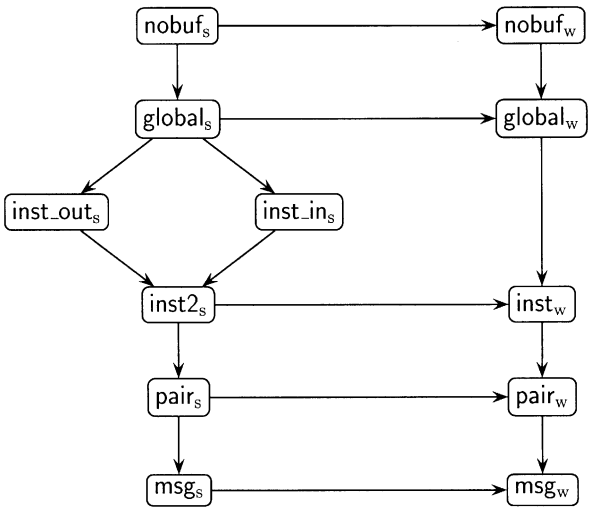

Fig. 19. Incomplete hierarchy.

strong and weak implementability it is clear that any X_s-implementable MSC is also X_w-implementable. The remaining classes are ordered as shown in Fig. 19.

An arrow pointing from one of the classes to another means that all MSCs that are implementable in the communication model corresponding to the first class are also implementable in the communication model corresponding to the second class. Any superfluous arrows (those that can be inferred from the transitivity of the relation) have been removed.

These evident relationships between the two hierarchies have led us to the further investigation of such relationships. As it turns out there are more relationships between and identifications of the classes from the two hierarchies. First, we prove that some classes can be identified.

Lemma 27. *An MSC is* pair_s*-implementable if and only if it is* pair_w*-implementable.*

Proof. Clearly, any pair_s-implementable MSC is also pair_w-implementable. It remains to prove that any pair_w-implementable MSC is also pair_s-implementable. Let k be a pair_w-implementable MSC. Let t be an arbitrary msc-trace of k. Let $m, m' \in M$ such that $\mathsf{from}(m) = \mathsf{from}(m')$ and $\mathsf{to}(m) = \mathsf{to}(m')$. We want to prove that $!m <_t^{\mathsf{trace}} !m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m'$, from which it follows that the (arbitrary) trace t is pair-implementable.

Suppose that $!m <_t^{\mathsf{trace}} !m'$. Then, because $\mathsf{from}(m) = \mathsf{from}(m')$ and $!m <_t^{\mathsf{trace}} !m'$, we have $!m <_k^{\mathsf{msc}} !m'$ (when $\mathsf{from}(m) = \mathsf{from}(m')$, either $!m <_t^{\mathsf{msc}} !m'$ or $!m' <_t^{\mathsf{msc}} !m$, and the second cannot be the case). Since k is pair_w-implementable there exists a trace t' that is pair-implementable. Since $!m <_k^{\mathsf{msc}} !m'$ we have $!m <_{t'}^{\mathsf{trace}} !m'$. Since t' is pair-implementable we have by Lemma 16 that $?m <_{t'}^{\mathsf{trace}} ?m'$. Because $\mathsf{to}(m) = \mathsf{to}(m')$ we then have $?m <_k^{\mathsf{msc}} ?m'$. Therefore we have $?m <_t^{\mathsf{trace}} ?m'$, which completes the proof. □

Lemma 28. *An MSC is* msg_s*-implementable if and only if it is* msg_w*-implementable.*

Proof. Trivial, because every impl-trace is msg-implementable, and thus each msc-trace is as well. □

Lemmas 27 and 28 establish that the classes pair_s and pair_w, and msg_s and msg_w are equivalent. In the remainder we denote these by pair and msg, respectively.

Next, we will prove that any $\mathsf{inst\_out}_s$-implementable MSC is global_w-implementable and that any $\mathsf{inst\_in}_s$-implementable MSC is global_w-implementable. To do this we first give some alternative characterisations for these implementations.

Lemma 29. *An MSC* k *is* $\mathsf{inst\_out}_s$*-implementable if and only if* $<_k^{\mathsf{io}} = <_k^{\mathsf{msc}}$. *An MSC* k *is* $\mathsf{inst\_in}_s$*-implementable if and only if* $<_k^{\mathsf{ii}} = <_k^{\mathsf{msc}}$.

Proof. We only give the proof for the first proposition. The proof of the second proposition follows the same lines.

First, suppose that MSC k is $\mathsf{inst\_out}_s$-implementable. By definition $<_k^{\mathsf{msc}} \subseteq <_k^{\mathsf{io}}$, so it only remains to be proven that $<_k^{\mathsf{io}} \subseteq <_k^{\mathsf{msc}}$. Suppose that $e <_k^{\mathsf{io}} e'$ for arbitrary $e, e' \in E_{\mathsf{msc}}(M)$. Then we have the existence of $e_1, \ldots, e_n$ such that $e \equiv e_1$, $e' \equiv e_n$ and for all $1 \leqslant i < n$ we have one of the following:

- $e_i <_k^{\mathsf{msc}} e_{i+1}$;
- $e_i \equiv ?m$ and $e_{i+1} \equiv ?m'$ for some $m, m' \in M$ such that $\mathsf{from}(m) = \mathsf{from}(m')$ and $!m <_k^{\mathsf{msc}} !m'$.

In the second case we have, by Lemma 6, $!m <_t^{\mathsf{trace}} !m'$ for every msc-trace t of k. Since k is $\mathsf{inst\_out}_s$-implementable we have that every msc-trace of k is inst_out-implementable. Thus, by Lemma 16 and the assumption that $\mathsf{from}(m) = \mathsf{from}(m')$ we have $?m <_t^{\mathsf{trace}} ?m'$ for every msc-trace t of k. Then, again by Lemma 6, we have $?m <_k^{\mathsf{msc}} ?m'$. In the first case we already know that $e_i <_k^{\mathsf{msc}} e_{i+1}$, and taking all these steps together we have $e <_k^{\mathsf{msc}} e'$, from which it follows that $<_k^{\mathsf{io}} \subseteq <_k^{\mathsf{msc}}$.

Second, suppose that $<_k^{\mathsf{io}} = <_k^{\mathsf{msc}}$. Then we must prove that MSC k is $\mathsf{inst\_out}_s$-implementable. Let t be an msc-trace of k, and let $m, m' \in M$ such that $\mathsf{from}(m) = \mathsf{from}(m')$. Suppose $!m <_t^{\mathsf{trace}} !m'$. Then because of $\mathsf{from}(m) = \mathsf{from}(m')$, we have $!m <_k^{\mathsf{msc}} !m'$. By the definition of $<_k^{\mathsf{io}}$ we then have $?m <_k^{\mathsf{io}} ?m'$. By the assumption that $<_k^{\mathsf{io}} = <_k^{\mathsf{msc}}$, this implies $?m <_k^{\mathsf{msc}} ?m'$, and thus $?m <_t^{\mathsf{trace}} ?m'$. □

For a similar characterisation of global_w-implementability we define a relation $<_k^{\mathsf{g}}$.

Definition 30. Let k be an MSC. The relation $<_k^{\mathsf{g}}$ on $E_{\mathsf{msc}}(M)$ is defined as the smallest relation that satisfies:

(1) $<_k^{\mathsf{msc}} \subseteq <_k^{\mathsf{g}}$;
(2) $<_k^{\mathsf{g}}$ is transitive;
(3) $!m <_k^{\mathsf{g}} !m' \Leftrightarrow ?m <_k^{\mathsf{g}} ?m'$ for all $m, m' \in M$.

Lemma 31. *An MSC* k *is* global_w*-implementable if and only if the relation* $<_k^{\mathsf{g}}$ *is cycle-free.*

Proof. First, suppose that MSC k is global_w-implementable. Let t be a global-implementable msc-trace of k. Then $<_t^{\mathsf{trace}}$ adheres to the restrictions in Definition 30, and thus $<_k^{\mathsf{g}} \subseteq <_t^{\mathsf{trace}}$, and $<_k^{\mathsf{g}}$ is cycle-free.

Second, suppose that the relation $<_k^{\mathsf{g}}$ is cycle-free. The idea of the proof is that we extend this relation until it is a total order. Then, if we can prove that the msc-trace corresponding with this total order is global-implementable, we are done.

We extend the relation $<_k^{\mathsf{g}}$ to form an order $<$ by repeatedly choosing a smallest element that has not yet been chosen, and taking that as the next element of our total order, all the while ensuring that the preconditions of Definition 30 are still being met. More formally, we will use the following algorithm (with S and $<$ as variables):

(1) $S := E_{\mathsf{msc}}(M)$, $< := <_k^{\mathsf{g}}$.
(2) Let e be any smallest element of S with respect to $<$, that is, any element of S for which there is no $e' \in S$ with $e' < e$.
(3) $S := S \backslash \{e\}$
(4) $< := (< \cup \{(e, e') \mid e' \in S\})^+$
(5) if $e \equiv !m$ for some $m \in M$, then $< := (< \cup \{(?m, ?m') \mid !m' \in S\})^+$
(6) Repeat steps 2 to 5 until $S = \emptyset$.

We first remark that the following invariant holds: $?m < ?m' \Rightarrow ?m <_k^{\mathsf{g}} ?m' \vee !m \notin S$ for all $m, m' \in M$. This clearly holds at the beginning, and only pairs $(?m, ?m')$ are added for which $!m \notin S$ since, otherwise, $?m$ would not be a smallest element of S. Also, after every execution of the body of the repetition (i.e. after step 5), $<$ is a total order on those events that are not contained in S.

Before we can make any arguments regarding the resulting order $<$, we have to prove that the algorithm is well-defined. In particular, for step 2 of the above algorithm it is necessary that $<$ is cycle-free. After step 1 $<$ is cycle-free because by the assumption $<_k^{\mathsf{g}}$ is cycle-free. There are two places where the relation $<$ is extended, namely step 4 and step 5. Step 4 maintains cycle-freeness of $<$. This can be seen as follows. Let e be an arbitrary smallest element of S with respect to $<$. Suppose that by adding the pairs (e, e') for $e' \in S \backslash \{e\}$ to $<$ a cycle appears. Then $e' < e$ for some $e' \in S \backslash \{e\}$ which contradicts the assumption that e is a smallest element of S with respect to $<$.

Step 5 maintains cycle-freeness as-well. Suppose that $!m$ is a smallest element of $<$ with respect to S. Suppose that a cycle is introduced by step 5. This can only be the case if a pair $(?m, ?m')$ is added to $<$ for which we already had $?m' < ?m$ and $!m' \in S$. By the previously mentioned invariant we have $?m' <_k^{\mathsf{g}} ?m$. By the definition of $<_k^{\mathsf{g}}$ then also $!m' <_k^{\mathsf{g}} !m$. As $!m' \in S$ this contradicts the assumption that $!m$ was a smallest element of S with respect to $<$. Thus, we have established that step 2 of the algorithm is well-defined. The other steps cause no problems, so the algorithm is well-defined.

The algorithm is guaranteed to terminate as the number of elements of the finite set S is decreased by one every time the body of the repetition is executed. Furthermore, because $<$ is a total order on those events that are not contained in S, and S is empty when the algorithm ends, upon termination of the algorithm, $<$ is a total order on $E_{\mathsf{msc}}(M)$. This total order corresponds to an msc-trace of the MSC as $<_k^{\mathsf{msc}} \subseteq <_k^{\mathsf{g}} \subseteq <$.

All that remains to be proven is that $<$ corresponds to a global-implementable msc-trace of k. Note that, after step 1, for all $m, m' \in M$ we have $!m < !m' \Rightarrow ?m < ?m'$. If in

step 4, $!m < !m'$ is added then in step 5 $?m < ?m'$ is added. Thus, $!m < !m' \Rightarrow ?m < ?m'$ is an invariant, from which it follows that the msc-trace corresponding with $<$ is global-implementable. □

Lemma 32. *If $e <_k^{\mathsf{g}} e'$, there is a sequence of events $e_1 e_2 \ldots e_n$, such that*:

(1) $e \equiv e_1$, $e' \equiv e_n$.
(2) *Either $e_i <_k^{\mathsf{msc}} e_{i+1}$ or $e_i \equiv ?m$ and $e_{i+1} \equiv !m$ for a certain m (for each i such that $1 \leqslant i < n$).*
(3) *The number of e_i's for which $e_i \not<_k^{\mathsf{msc}} e_{i+1}$ (and thus $e_i \equiv ?m$ and $e_{i+1} \equiv !m$ hold) is less than or equal to the number of e_i's for which $e_i \equiv !m$ and $e_{i+1} \equiv ?m$.*

Thus the sequence consists of $<^{\mathsf{msc}}$-orderings with additionally some messages that are passed 'in the wrong direction', but there are at least as many messages passed in the right as in the wrong direction.

As an example, look at MSC 2*a* in Fig. 11. In this MSC, $!a <^{\mathsf{g}} !b$. The sequence of events corresponding to the lemma is $!a\,?a\,?b\,!b$. There is one message (b) that is passed from receipt to sending, and one message (a) that is passed from sending to receipt.

Proof. In this proof we will denote the sequence $e \equiv e_1, \ldots, e_n \equiv e'$ for a given e and e' by $\overrightarrow{(e, e')}$. This sequence is of course in general not uniquely defined, but this does not matter for the proof.

First we note that $<_k^{\mathsf{g}}$ can be constructed by the following algorithm:

(1) $<_k^{\mathsf{g}} := <_k^{\mathsf{msc}}$
(2) $<_k^{\mathsf{g}} := <_k^{\mathsf{g}} \cup \{(?m, ?m') \mid !m <_k^{\mathsf{g}} !m'\}$
(3) $<_k^{\mathsf{g}} := <_k^{\mathsf{g}} \cup \{(!m, !m') \mid ?m <_k^{\mathsf{g}} ?m'\}$
(4) $<_k^{\mathsf{g}} := <_k^{\mathsf{g}} \cup \{(e, e') \mid \exists e'' : e <_k^{\mathsf{g}} e'' \wedge e'' <_k^{\mathsf{g}} e'\}$
(5) Repeat steps 2 to 4 until no change occurs.

We will prove that the lemma remains true throughout the running of this algorithm. It is trivially true after step 1. Suppose that step 2 introduces a new pair into $<_k^{\mathsf{g}}$, $?m <_k^{\mathsf{g}} ?m'$. Then $!m <_k^{\mathsf{g}} !m'$ already is part of $<_k^{\mathsf{g}}$, so by induction hypothesis $\overrightarrow{(!m, !m')}$ exists. Then

$$\overrightarrow{(?m, ?m')} = (?m) +\!\!+ \overrightarrow{(!m, !m')} +\!\!+ (?m')$$

(where $(e_1, \ldots, e_n) +\!\!+ (f_1, \ldots, f_n)$ is defined to be $(e_1, \ldots, e_n, f_1, \ldots, f_n)$) satisfies the requirements. There is one pair of the form $(?m, !m)$ added, but also one of the form $(!m', ?m')$, so this step does not invalidate the lemma. Likewise, if step 3 introduces a new pair $!m <_k^{\mathsf{g}} !m'$, we can choose $\overrightarrow{(!m, !m')} = (!m) +\!\!+ \overrightarrow{(?m, ?m')} +\!\!+ (!m')$. Finally, if step 4 introduces a new pair $e <_k^{\mathsf{g}} e'$, we can choose $\overrightarrow{(e, e')} = \overrightarrow{(e, e'')} +\!\!+ \overrightarrow{(e'', e')}$ (or rather, we should remove one of the now double e'' to get a correct sequence). □

Lemma 33. *Every* $\mathsf{inst\_out_s}$*-implementable MSC is* $\mathsf{global_w}$*-implementable. Every* $\mathsf{inst\_in_s}$*-implementable MSC is* $\mathsf{global_w}$*-implementable.*

Proof. We prove this for an $\mathsf{inst\_out_s}$-implementable MSC. The proof is completely analogous for a $\mathsf{inst\_in_s}$-implementable MSC.

We prove this by contradiction, so we assume that k is an $\mathsf{inst\_out_s}$-implementable MSC that is not $\mathsf{global_w}$-implementable. By Lemma 29 we have $<_k^{\mathsf{io}} = <_k^{\mathsf{msc}}$, and by Lemma 31 we have that $<_k^{\mathsf{g}}$ has a cycle.

Because $<_k^{\mathsf{g}}$ has a cycle, we can conclude from Lemma 32 that there is a cycle of steps which are either steps of $<_k^{\mathsf{msc}}$ or of the form $(?m, !m)$, where, furthermore, the number of steps of the form $(?m, !m)$ is not greater than the number of steps of the form $(!m, ?m)$. We call such a cycle a quasi-cycle of order N, where N is the number of times that a step of the form $(?m, !m)$ occurs in the cycle.

We prove that this cycle can be changed into a quasi-cycle of order 0. Let the order be greater than 0. Because the quasi-cycle is a cycle, and contains at least one $(?m, !m)$-step and at least one $(!m, ?m)$-step, there will be at least one $(!m, ?m)$-step, such that after that $(!m, ?m)$-step a $(?m, !m)$-step will take place before the next $(?m, !m)$-step. Thus, the quasi-cycle contains a subsequence $(?m, !m, \ldots, !m', ?m')$, where there are no steps of the forms $(?m, !m)$ or $(!m, ?m)$ between $!m$ and $!m'$.

Because we have $!m <_k^{\mathsf{msc}} !m'$, by definition we get $?m <_k^{\mathsf{io}} ?m'$, from which we get $?m <_k^{\mathsf{msc}} ?m'$ from the assumption that $<_k^{\mathsf{msc}} = <_k^{\mathsf{io}}$. Thus, by removing all steps between $?m$ and $?m'$, and replacing them with a single step, we still have a cycle of $<_k^{\mathsf{msc}}$ and $(!m, ?m)$ steps, but with one less occurrence of both the type $(!m, ?m)$ and the type $(?m, !m)$. Thus, this is a quasi-cycle of order $N - 1$. Repeating this, we will finally obtain a quasi-cycle of order 0. However, a quasi-cycle of order zero is a cycle of only $<^{\mathsf{msc}}$-steps.

Thus, we see that, given the assumption, $<^{\mathsf{msc}}$ must have a cycle. This is impossible, so the assertions cannot simultaneously hold, so each $\mathsf{inst\_out_s}$-implementable MSC is $\mathsf{global_w}$-implementable. □

In Fig. 20 we give all communication models that remain after the identifications obtained until now. The arrows between these models follow also from the previous theorems and lemmas. Finally, we have to prove that the arrows between models from the strong and weak hierarchy are strict and that there are no additional arrows necessary. It suffices to show that the following arrows do not exist: $\mathsf{global_s}$ to $\mathsf{nobuf_w}$, $\mathsf{nobuf_w}$ to $\mathsf{inst2_s}$, and $\mathsf{inst2_s}$ to $\mathsf{global_w}$. The rest then follows because of transitivity. For example, the non-existence of an arrow from $\mathsf{global_s}$ to $\mathsf{nobuf_w}$ implies the non-existence of an arrow from $\mathsf{inst\_out_s}$ to $\mathsf{nobuf_w}$, because if the second arrow exists then, by transitivity, also the first must exist. Similarly we obtain the non-existence of arrows from $\mathsf{inst\_in_s}$ and $\mathsf{inst2_s}$ to $\mathsf{nobuf_w}$. We use the MSCs in Figs. 21 and 22 to indicate that the first two arrows do not exist. MSC 7 is $\mathsf{global_s}$-implementable, but not $\mathsf{nobuf_w}$-implementable. It has one trace, $!a\,!b\,?a\,?b$, which is global-implementable, but not nobuf-implementable. We see that MSC 7 contains only one instance, so all messages are messages to the same instance that sent them.

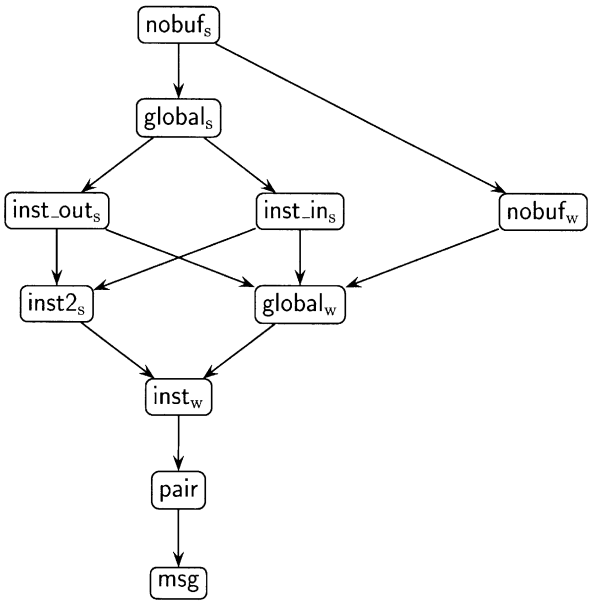

Fig. 20. Final hierarchy.

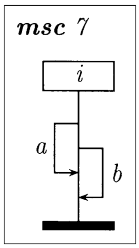

Fig. 21. MSC distinguishing global$_s$- and nobuf$_w$-implementability.

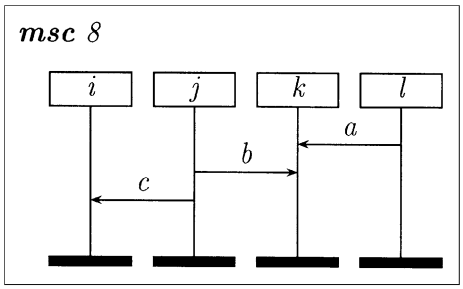

Fig. 22. MSC distinguishing inst2$_s$- and nobuf$_w$-implementability.

This is no coincidence, it can be shown that all possible counterexamples have such messages.

MSC 8 is nobuf_w-implementable, but not $\mathsf{inst2}_s$-implementable. That it is nobuf_w-implementable can be seen from the picture, which shows that there is the msc-trace $!a\,?a\,!b\,?b\,!c\,?c$, which is nobuf-implementable. However, the trace $!b\,!c\,?c\,!a\,?a\,?b$ is not inst2-implementable: Because $?b$ is after $?a$ in the trace, $!!b$ must be after $!!a$ to make the trace inst-input implementable, while, because $!b$ is before $!c$, $!!b$ must be before $!!c$ to make the trace inst-output implementable. However, $!!a$ must be after $!a$ and $!!c$ before $?c$, so $!!c$ will be before $!!a$ in any extension of this trace, which implies that $!!b$ cannot be both before $!!c$ and after $!!a$.

The non-existence of an arrow from $\mathsf{inst2}_s$ to global_w is taken care of by MSC 6 in Fig. 17. It has already been shown not to be global_w-implementable. It is $\mathsf{inst2}_s$-implementable because every msc-trace of this MSC can be extended to an inst2-implementable impl-trace by adding $!!a$ and $!!b$ immediately after $!a$ and $!b$, and $!!c$ and $!!d$ immediately before $?c$ and $?d$.

Theorem 34. *The communication models from Fig.* 20 *are all different, and they are ordered as expressed in Fig.* 20.

Proof. This has been explained in the text above. □

5. Characterisations

Thus far, we have considered the notions of strong and weak implementability and we have ordered those in a hierarchy. In this section, we will consider how to determine the implementability of a given MSC with respect to a given communication model. That is, we study the algorithmic aspects of the communication models. The original definitions of the communication models are hard to check automatically. To do so would require one to look at all msc-traces, possibly even all impl-traces, of the MSC and check whether or not they are implementable with respect to the communication model. An MSC can have many traces, in fact their number is exponential in the number of events of the MSC.

In the previous sections, for the communication models global_w, global_s, inst_w, $\mathsf{inst\_out}_s$, and $\mathsf{inst\_in}_s$ characterisations have already been given that are easier to check. These are based on cycle-freeness of relations between the events, or the equality of two orders. Both the creation of these relations and orders, and checking for their cycle-freeness or equality can be done in polynomial time in the number of events. For the communication models pair and msg the fact that weak and strong implementability coincide leads directly to an easy to use characterisation: Because implementability of a single msc-trace and implementability of all msc-traces are equivalent, looking at one single trace suffices. Thus, we only need new characterisations for nobuf_w, nobuf_s, and $\mathsf{inst2}_s$.

Definition 35. Let k be an MSC over the set of messages M. The relation $<_k^w$ on M is for all $m, m' \in M$ defined by $m <_k^w m'$ if and only if $!m <_k^{msc} ?m'$ and $m \not\equiv m'$.

Lemma 36. *An MSC k is* $\mathsf{nobuf_w}$*-implementable if and only if the relation* $<_k^{\mathsf{w}}$ *is cycle-free.*

Proof. Let k be an MSC over the set of messages M. First, suppose that k is $\mathsf{nobuf_w}$-implementable. Suppose furthermore that $<_k^{\mathsf{w}}$ has a cycle, say $m_1 <_k^{\mathsf{w}} m_2 <_k^{\mathsf{w}} \cdots <_k^{\mathsf{w}} m_n$ for some $m_1, m_2, \ldots, m_n \in M$ such that $m_1 \equiv m_n$. Then, from the definition of $<_k^{\mathsf{w}}$ and $<_k^{\mathsf{msc}}$, we obtain for all $1 \leqslant i < n$ that $!m_i <_k^{\mathsf{msc}} ?m_{i+1}$ and $!m_{i+1} <_k^{\mathsf{msc}} ?m_{i+1}$. Then, for every msc-trace t of k, we must have $!m_i <_t^{\mathsf{trace}} ?m_{i+1}$ and $!m_{i+1} <_t^{\mathsf{trace}} ?m_{i+1}$ for all $1 \leqslant i < n$. Since k is $\mathsf{nobuf_w}$-implementable, there is a nobuf-implementable msc-trace t'. In this trace, there can be no events between $!m_{i+1}$ and $?m_{i+1}$, so $!m_i <_{t'}^{\mathsf{trace}} ?m_{i+1}$ implies $!m_i <_{t'}^{\mathsf{trace}} !m_{i+1}$. Thus we get $!m_1 <_{t'}^{\mathsf{trace}} !m_2 <_{t'}^{\mathsf{trace}} \cdots <_{t'}^{\mathsf{trace}} !m_n$ and since $!m_1 \equiv !m_n$ we thus have a cycle of $<_{t'}^{\mathsf{trace}}$. Thus such a nobuf-implementable msc-trace t' does not exist. This contradicts the assumption that k is $\mathsf{nobuf_w}$-implementable. Therefore, $<_k^{\mathsf{w}}$ is cycle-free.

Second, suppose that $<_k^{\mathsf{w}}$ is cycle-free. We extend $<_k^{\mathsf{w}}$ to a total order $<$, say $m_1 < m_2 < \cdots < m_n$ where $M = \{m_1, m_2, \ldots, m_n\}$. Then the trace

$$t \equiv !m_1\, ?m_1\, !m_2\, ?m_2 \cdots !m_n\, ?m_n$$

is clearly nobuf-implementable. Thus, if suffices to prove that the trace t is a trace of MSC k. Thereto, suppose that $e <_k^{\mathsf{msc}} e'$ for some $e, e' \in E_{\mathsf{msc}}(M)$. We distinguish four cases:

- $e \equiv !m$ and $e' \equiv !m'$ for some $m, m' \in M$. As $!m <_k^{\mathsf{msc}} !m'$ and $!m' <_k^{\mathsf{msc}} ?m'$, we also have $!m <_k^{\mathsf{msc}} ?m'$. Then, by the definition of $<_k^{\mathsf{w}}$, we have $m <_k^{\mathsf{w}} m'$, and therefore $!m <_t^{\mathsf{trace}} !m'$.
- $e \equiv !m$ and $e' \equiv ?m'$ for some $m, m' \in M$. If $m \equiv m'$, then trivially $!m <_t^{\mathsf{trace}} ?m'$. Otherwise, by the definition of $<_k^{\mathsf{w}}$, we have $m <_k^{\mathsf{w}} m'$, and therefore $!m <_t^{\mathsf{trace}} ?m'$.
- $e \equiv ?m$ and $e' \equiv !m'$ for some $m, m' \in M$. As $!m <_k^{\mathsf{msc}} ?m$, $?m <_k^{\mathsf{msc}} !m'$ and $!m' <_k^{\mathsf{msc}} ?m'$, we have $!m <_k^{\mathsf{msc}} ?m'$. Then, by the definition of $<_k^{\mathsf{w}}$, we have $m <_k^{\mathsf{w}} m'$, and therefore $?m <_t^{\mathsf{trace}} !m'$.
- $e \equiv ?m$ and $e' \equiv ?m'$ for some $m, m' \in M$. As $!m <_k^{\mathsf{msc}} ?m$ and $?m <_k^{\mathsf{msc}} ?m'$, we have $!m <_k^{\mathsf{msc}} ?m'$. Then, by the definition of $<_k^{\mathsf{w}}$, we have $m <_k^{\mathsf{w}} m'$, and therefore $?m <_t^{\mathsf{trace}} ?m'$.

In each of the four cases we have $e <_t^{\mathsf{trace}} e'$, which completes the proof. □

Lemma 37. *If an MSC k is* $\mathsf{nobuf_s}$*-implementable, then* $<_k^{\mathsf{msc}}$ *is a total order.*

Proof. Let k be an MSC over the set of messages M. We use contraposition, so assuming that $<_k^{\mathsf{msc}}$ is not a total order, we prove that k is not $\mathsf{nobuf_s}$-implementable. Let t be an arbitrary msc-trace of the MSC. Because $<_k^{\mathsf{msc}}$ is not a total order, there are events $e, e' \in E_{\mathsf{msc}}(M)$ such that $e <_t^{\mathsf{trace}} e'$, but not $e <_k^{\mathsf{msc}} e'$. For any event $e'' \in E_{\mathsf{msc}}(M)$ with $e <_t^{\mathsf{trace}} e'' <_t^{\mathsf{trace}} e'$ we have either $e \not<_k^{\mathsf{msc}} e''$ or $e'' \not<_k^{\mathsf{msc}} e'$ as otherwise $e <_k^{\mathsf{msc}} e'$. So there also is a such a pair of events that are immediately after one another in the msc-trace t. Then, interchanging these events would result in another msc-trace t' of the MSC. It cannot be the case that both t and t' are nobuf-implementable. □

Lemma 38. *Let $<$ be a partial order, such that $b \not< a$ and $d \not< c$, and let $<' = (< \bigcup \{(a,b),(c,d)\})^+$ contain a cycle. Then it contains a simple cycle with both (a,b) and (c,d) part of this cycle.*

Proof. If $<'$ has a cycle, then so does $< \bigcup \{(a,b),(c,d)\}$. Look at an arbitrary simple cycle of $< \bigcup \{(a,b),(c,d)\}$. If this cycle did not contain (a,b) or (c,d), then this would also be a cycle of $<$. If it contained (a,b) but not (c,d), we would have $b < a$, and if it contained (c,d) but not (a,b), we would have $d < c$. Thus, the cycle must contain both (a,b) and (c,d). □

Lemma 39. *An MSC k is* global_s*-implementable if and only if for all $m, m' \in M$, we either have both $!m <_k^{\mathsf{msc}} !m'$ and $?m <_k^{\mathsf{msc}} ?m'$, or we have both $!m' <_k^{\mathsf{msc}} !m$ and $?m' <_k^{\mathsf{msc}} ?m$.*

Proof. First, suppose that MSC k is global_s-implementable. Let $m, m' \in M$. Without loss of generality, we may assume $!m' \not<_k^{\mathsf{msc}} !m$. Then it suffices to prove that $!m <_k^{\mathsf{msc}} !m'$ and $?m <_k^{\mathsf{msc}} ?m'$. Now we can distinguish two cases: $!m <_k^{\mathsf{msc}} !m'$ and $!m \not<_k^{\mathsf{msc}} !m'$.

Suppose that $!m <_k^{\mathsf{msc}} m'$. Then, by Lemma 6, $!m <_t^{\mathsf{trace}} !m'$ for every msc-trace t of k. Since every msc-trace of k is global-implementable, we have by Lemma 16 that $?m <_t^{\mathsf{trace}} ?m'$ for every msc-trace t of k. Then, again by Lemma 6, we have $?m <_k^{\mathsf{msc}} ?m'$, which completes this part of the proof.

Now, suppose that $!m \not<_k^{\mathsf{msc}} !m'$. A similar reasoning as above shows that $?m <_k^{\mathsf{msc}} ?m'$ implies $!m <_k^{\mathsf{msc}} !m'$ (remember that the single arrow in Lemma 16 is allowed to be read as a double arrow), so $?m \not<_k^{\mathsf{msc}} ?m'$, and analogously $?m' \not<_k^{\mathsf{msc}} ?m$. We will now show that there exists an msc-trace t of k such that $!m <_t^{\mathsf{trace}} !m'$ and $?m' <_t^{\mathsf{trace}} ?m$, thereby contradicting the assumption that k is global_s-implementable. We define the order $<$ as follows: $< = (<_k^{\mathsf{msc}} \cup \{(!m, !m'), (?m', ?m)\})^+$. We prove that $<$ is cycle-free, from which it immediately follows that there is an msc-trace t such that $!m <_t^{\mathsf{trace}} !m'$ and $?m' <_t^{\mathsf{trace}} ?m$ (just extend $<$ to a total order). Assume that $<$ is not cycle-free. From Lemma 38 we can conclude that there exists a simple cycle in $<$ with both $!m < !m'$ and $?m' < ?m$. Because this is a simple cycle, this would imply that $!m < !m' <_k^{\mathsf{msc}} ?m' < ?m <_k^{\mathsf{msc}} !m$. However, $?m <_k^{\mathsf{msc}} !m$ is impossible because $!m <_k^{\mathsf{msc}} ?m$ and $<_k^{\mathsf{msc}}$ is cycle-free. Thus, $<$ is cycle-free, which leads to a contradiction with the assumption that k is global_s-implementable, so this possibility cannot occur.

Second, suppose that for all $m, m' \in M$ we have $!m <_k^{\mathsf{msc}} !m'$ and $?m <_k^{\mathsf{msc}} ?m'$, or $!m' <_k^{\mathsf{msc}} !m$ and $?m' <_k^{\mathsf{msc}} ?m$. We must prove that MSC k is global_s-implementable. Let t be an arbitrary msc-trace of MSC k. Let $m, m' \in M$. By Lemma 16 it suffices to prove that $!m <_t^{\mathsf{trace}} !m' \Rightarrow ?m <_t^{\mathsf{trace}} ?m'$. Suppose that $!m <_t^{\mathsf{trace}} !m'$. Then, by Lemma 6, $!m' \not<_k^{\mathsf{msc}} !m$. Therefore, by the assumption, $!m <_k^{\mathsf{msc}} !m'$ and $?m <_k^{\mathsf{msc}} ?m'$. So, by Lemma 6, we have $?m <_t^{\mathsf{trace}} ?m'$. □

Lemma 40. *An MSC k is* $\mathsf{inst2}_s$*-implementable if and only if $<_k^{\mathsf{i2}} = <_k^{\mathsf{impl}}$.*

Proof. Let k be an MSC over the set of messages M. First, suppose that k is $\mathsf{inst2_s}$-implementable. By definition, $<_k^{\mathsf{impl}} \subseteq <_k^{\mathsf{i2}}$, so it only remains to be proven that $<_k^{\mathsf{i2}} \subseteq <_k^{\mathsf{impl}}$. Suppose that $e <_k^{\mathsf{i2}} e'$ for some $e, e' \in E_{\mathsf{impl}}(M)$. Then we have the existence of $e_1, \ldots, e_n$ such that $e \equiv e_1$, $e' \equiv e_n$ and for all $1 \leqslant i < n$ we have one of the following:

- $e_i <_k^{\mathsf{impl}} e_{i+1}$;
- $e_i \equiv !!m$ and $e_{i+1} \equiv !!m'$ for some $m, m' \in M$ such that $\mathsf{from}(m) = \mathsf{from}(m')$ and $!m <_k^{\mathsf{impl}} !m'$;
- $e_i \equiv !!m$ and $e_{i+1} \equiv !!m'$ for some $m, m' \in M$ such that $\mathsf{to}(m) = \mathsf{to}(m')$ and $?m <_k^{\mathsf{impl}} ?m'$.

In the second case we use induction on the number of output events $!m''$ that can be in between $!m$ and $!m'$ to prove that $!!m <_k^{\mathsf{impl}} !!m'$.

- If there is no output event $!m''$ such that $!m <_k^{\mathsf{impl}} !m'' <_k^{\mathsf{impl}} !m'$, then either $!m <^{\mathsf{inst}} !m'$ or $?m <_k^{\mathsf{impl}} !m'$. In the first case, if $!!m <_k^{\mathsf{impl}} !!m'$ did not hold, $<_k^{\mathsf{impl}} \cup \{(!!m', !!m)\}$ would be cycle-free. Any extension of this relation to a total order would be $<_t^{\mathsf{trace}}$ for an impl-trace t that is not $\mathsf{inst\_out}$-implementable, and thus not $\mathsf{inst2}$-implementable. In the second case we have $!!m <_k^{\mathsf{impl}} ?m <_k^{\mathsf{impl}} !m' <_k^{\mathsf{impl}} !!m'$.
- If there is at least one output event $!m''$ such that $!m <_k^{\mathsf{impl}} !m'' <_k^{\mathsf{impl}} !m'$, then, using the induction hypothesis, we have $!!m <_k^{\mathsf{impl}} !!m'' <_k^{\mathsf{impl}} !!m'$.

For the third case a similar reasoning gives $e_i <_k^{\mathsf{impl}} e_{i+1}$. Thus, in all cases we obtain $e_i <_k^{\mathsf{impl}} e_{i+1}$ and therefore also $e <_k^{\mathsf{impl}} e'$ which was to be proven.

Second, suppose that $<_k^{\mathsf{i2}} = <_k^{\mathsf{impl}}$. Let t be an impl-trace of k, and let $m, m' \in M$. Suppose that $\mathsf{from}(m) = \mathsf{from}(m')$ and that $!m <_t^{\mathsf{trace}} !m'$. Then we have that $!m <_t^{\mathsf{trace}} !m'$ implies $!m <_k^{\mathsf{impl}} !m'$. Then, by the definition of $<^{\mathsf{i2}}$, we have $!!m <_k^{\mathsf{i2}} !!m'$. Since we assumed that $<_k^{\mathsf{i2}} = <_k^{\mathsf{impl}}$ we also have $!!m <_k^{\mathsf{impl}} !!m'$, and therefore $!!m <_t^{\mathsf{trace}} !!m'$.

Reversely, suppose that $!m \not<_t^{\mathsf{trace}} !m'$. With a fully analogous argument we obtain $!!m \not<_t^{\mathsf{trace}} !!m'$. The proof that $\mathsf{to}(m) = \mathsf{to}(m') \Rightarrow (?m <_k^{\mathsf{impl}} ?m' \Leftrightarrow !!m <^{\mathsf{impl}} !!m')$ is analogous. □

In the following theorem we list the characterisations for implementability we have given in this paper and we add characterisations for the implementabilities not yet characterised. An overview is presented in Fig. 23.

Theorem 41. (1) *An MSC k is* $\mathsf{nobuf_w}$*-implementable if and only if $<_k^{\mathsf{w}}$ is cycle-free.*

(2) *An MSC k is* $\mathsf{nobuf_s}$*-implementable if and only if it has exactly one msc-trace, and that trace is* nobuf*-implementable.*

(3) *An MSC k is* $\mathsf{global_s}$*-implementable if and only if for each pair of messages m and m' either both $!m <_k^{\mathsf{msc}} !m'$ and $?m <_k^{\mathsf{msc}} ?m'$, or both $!m' <_k^{\mathsf{msc}} !m$ and $?m' <_k^{\mathsf{msc}} ?m$ hold.*

(4) *An MSC k is* $\mathsf{global_w}$*-implementable if and only if $<_k^{\mathsf{g}}$ is cycle-free.*

(5) *An MSC k is* $\mathsf{inst\_out_s}$*-implementable if and only if $<_k^{\mathsf{io}} = <_k^{\mathsf{msc}}$.*

(6) *An MSC k is* $\mathsf{inst\_in_s}$*-implementable if and only if $<_k^{\mathsf{ii}} = <_k^{\mathsf{msc}}$.*

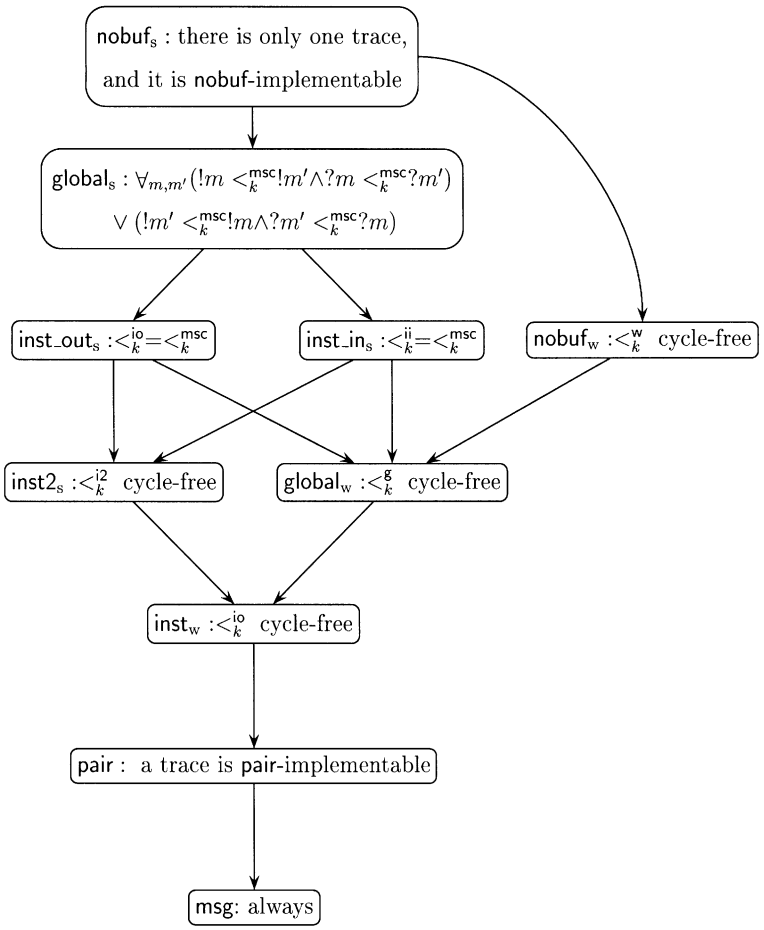

Fig. 23. Overview.

(7) *An MSC k is* inst2$_s$*-implementable if and only if* $<_k^{i2} = <_k^{impl}$.

(8) *An MSC k is* inst$_w$*-implementable if and only if* $<_k^{io}$ *is cycle-free.*

(9) *For any msc-trace t of an MSC k, k is* pair*-implementable if and only if t is* pair*-implementable.*

(10) *An MSC k is always* msg*-implementable.*

Proof. (1) See Lemma 36.

(2) If the MSC k is nobuf$_s$-implementable it has one trace because $<_k^{msc}$ is a total order (Lemma 37).

(3) See Lemma 39.

(4) See Lemma 31.

(5) See Lemma 29.

(6) See Lemma 29.

(7) See Lemma 40.
(8) See Lemma 23.
(9) First, if k is pair-implementable, it is pair_s-implementable and thus every trace t of k is pair-implementable. Second, if a randomly chosen trace t is pair-implementable, then k is pair_w-implementable, and thus pair_s-implementable.
(10) See Lemma 28. □

6. Related work

In this section we will compare our conclusions with those found in related literature.

In [6] Charron-Bost et al. discuss three different implementations for MSC-like diagrams: Realizable with Synchronous Communication (RSC), Causally Ordered (CO) and FIFO. They also define A (asynchronous), but this is (just like msg in our hierarchy) used to denote the set of all allowable diagrams, not some subset. They find that there is a strict inclusion $\mathrm{RSC} \subset \mathrm{CO} \subset \mathrm{FIFO} \subset A$.

Theorem 42. *The implementations that in* [6] *are named* RSC and FIFO *are equivalent to the implementations* nobuf_w, *and* pair. *The implementation* CO *is strictly between the implementations* inst_w *and* pair.

Proof.

- RSC-nobuf_w: Definition 3.6 in [6] states, after translating it into our terminology, that a computation is RSC if and only if there is a trace t for which for each $m \in M$ we have that the set $\{x \in C \mid !m <_t^{\mathsf{trace}} x <_t^{\mathsf{trace}} ?m\}$ is empty, which is equivalent to the definition that is obtained by combining Lemma 16 and Definition 20.
- FIFO-pair: The definition for FIFO in [6] (Definition 3.3) translates to (by rewriting the terminology of Charron-Bost et al. in ours): $!m <_k^{\mathsf{msc}} !m' \wedge \mathsf{from}(m) = \mathsf{from}(m') \wedge \mathsf{to}(m) = \mathsf{to}(m') \Rightarrow ?m <_k^{\mathsf{msc}} ?m'$, or $\mathsf{from}(m) = \mathsf{from}(m') \wedge \mathsf{to}(m) = \mathsf{to}(m') \Rightarrow (!m <_k^{\mathsf{msc}} !m' \Rightarrow ?m <_k^{\mathsf{msc}} ?m')$, which is seen to be equivalent to the definition in Lemma 16 once it is realised that (for the basic MSCs considered here) $\mathsf{to}(m) = \mathsf{to}(m') \Rightarrow (?m <_k^{\mathsf{msc}} ?m' \vee ?m' <_k^{\mathsf{msc}} ?m)$ and $\mathsf{from}(m) = \mathsf{from}(m') \Rightarrow (!m <_k^{\mathsf{msc}} !m' \vee !m' <_k^{\mathsf{msc}} !m)$.
- CO: That the class of pair-implementable MSCs is strictly greater than that of CO-implementable MSCs is shown in [6]. Remains to be shown that the class of CO-implementable MSCs is strictly greater than that of inst_w-implementable MSCs. Remains to be shown that the class of CO-implementable MSCs is strictly greater than that of inst_w-implementable MSCs. The definition of CO as given in [6] (Definition 3.4) can be translated to $\mathsf{to}(m) = \mathsf{to}(m') \wedge !m <_k^{\mathsf{msc}} !m' \Rightarrow ?m <_k^{\mathsf{msc}} ?m'$. An example of an MSC that is CO-implementable, but not inst_w-implementable, is the MSC *lobster* in Fig. 24. It is CO-implementable, because there is no pair of messages with $\mathsf{to}(m) = \mathsf{to}(m')$ where $!m$ and $!m'$ are ordered, but it is not inst_w-implementable, as can for example be seen by the fact that $!a <^{\mathsf{msc}} !c$ and $!b <^{\mathsf{msc}} !d$, and thus $?a <^{\mathsf{io}} ?c$ and $?b <^{\mathsf{io}} ?d$, while at the same time we clearly have $?c <^{\mathsf{io}} ?b$ and $?d <^{\mathsf{io}} ?a$, so $<^{\mathsf{io}}$ contains a cycle.

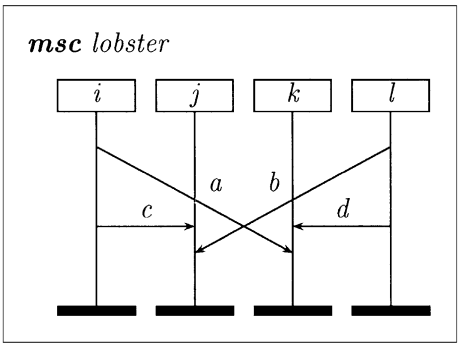

Fig. 24. MSC to distinguish CO- and inst_w-implementability.

It remains to be proven that each inst_w-implementable MSC is also CO-implementable. We do this using contraposition. So, let k be an MSC that is not CO-implementable. We then have that there are messages m and m' with $!m <_k^{\mathsf{msc}} !m'$, $?m \not<_k^{\mathsf{msc}} ?m'$ and $\mathsf{to}(m) = \mathsf{to}(m')$. From the last two facts we can derive that $?m' <^{\mathsf{msc}} ?m$, and thus we have both $?m <^{\mathsf{io}} ?m'$ (because $!m <_k^{\mathsf{msc}} !m'$) and $?m' <^{\mathsf{io}} ?m$ (because $?m' <^{\mathsf{msc}} ?m$), so the MSC k is not inst_w-implementable. □

Another paper in which different communication models for MSC have been studied, is [2]. Although some of their communication models are similar to some of ours, the works cannot be directly compared, because of the different focus. Whereas our question is whether an MSC can be implemented in a given communication model, their question is whether an MSC will run as expected if it is implemented on a given communication model.

Their main point of focus is the problem of race conditions, in which two messages which might be supposed by the user to be received in the order prescribed by the MSC, might in reality arrive in the reverse order. The communication model influences both which messages the user assumes to arrive in the correct order and which messages actually do.

This line of thought has been extended in [26]. There a set of channels is assumed, which can be any communication model between pair and msg (other possible models can be inserted easily, but were not looked at because of the specific subject of the paper, namely characterisation of an MSC in SDL). Then, for each message it is checked which messages on the same channel that have to be dealt with later may have been received earlier.

7. Concluding remarks and future research

We have considered communication models for asynchronous communication in Message Sequence chart. These models consist of FIFO buffers for the sending and reception of messages. By varying the locality of the buffers we have arrived, in a systematic way, at 25 models for communication. With respect to traces, consisting of

putting a message into a buffer and removing a message from a buffer, there are seven different models.

By lifting this implementability notion from traces to MSCs in two ways, strong and weak, we obtain fourteen models. After identification, ten essentially different models on the level of MSCs remain.

For defining the models we have used the notion of impl-traces; these are a natural extension of normal msc-traces if a message can pass two buffers on its way from source to destination.

In this paper, we have only considered basic message sequence charts. An interesting question is how to transfer the notions and properties defined for this simple language to the complete language MSC. As many of our theorems rely on the fact that the events on an instance are totally ordered, an extension to MSC with more sophisticated ordering mechanisms (e.g., coregion and causal ordering) will imply a revision of the hierarchy. Another interesting question is whether the implementation properties are preserved under composition by means of the operators of MSC.

Furthermore, we have restricted ourselves to the treatment of architectures in which each message has exactly one possible communication path and where each such path contains at most two buffers. The extension to more flexible architectures is non-trivial and is expected to lead to an extension of the hierarchy.

An important assumption that we have made in this paper, which is often not true in real-life examples, is the assumption of homogeneity, that is, the assumption that all instances have exactly the same type of buffers. In real life it may for example well be the case, that there is more than one channel between two instances, but some channels are still used for more than one message, thus creating an architecture somewhere between our 'one buffer per pair' and 'one buffer per message'. This subject has been given some attention in [2,26].

Finally, our assumption of infinite FIFO buffers may be relaxed, allowing other types of buffers and buffers with finite capacity.

The results obtained in this paper form a solid base for several applications. First, they allow us to discuss the relation between different variants of MSC, such as Interworkings [22]. Interworkings presuppose a synchronous communication mechanism. An Interworking can be considered as the restriction of the semantics of an MSC to only the nobuf-implementable traces. Thus, an MSC can be interpreted as an Interworking if and only if there is at least one such trace, i.e., the MSC is nobuf_w-implementable. This implies that using the theory in this paper, a formal semantics of Interworkings can be derived in a systematic way from the semantics of MSC. We also envisage tool-oriented applications. One could for example consider a tool in which a user can select a communication model, draw an MSC and invoke an algorithm to check if the MSC is implementable with respect to the selected model. Alternatively, the user can provide an MSC and use a tool to determine the minimal architecture, according to our hierarchy, which is needed for implementation.

Often, a user is interested in the question whether all msc-traces of his MSC are implementable with respect to a certain architecture. We can also envisage two possible uses relying on the implementability of a single msc-trace. First, MSCs are often used to display one single msc-trace, for example if it is the result of a simulation run. In this

case, the question is not whether the MSC is strongly or weakly implementable, but whether the implied msc-trace is implementable (as defined in Section 3). Second, given an MSC, a user may want to know if at least one msc-trace is implementable and if so, which trace that is. He is interested in a *witness*. Both applications can easily be derived from the results on weak implementability. The algorithms (see above) can easily be modified to check implementability of a given msc-trace and to produce a witness.

A more involved application would be to use a selected communication model to reduce the set of msc-traces defined by a given MSC to only those traces that are implementable on the given model. In this way, the semantics of an MSC would be relative to some selected model.

For most of these applications computer support would be useful. Based upon the definitions presented in this paper, it is feasible to derive efficient algorithms. All models in the weak spectrum can be characterised in terms of the cycle-freeness of an extended order relation, see Theorem 41. An example of such a characterisation is given in Theorem 23. There it is stated that an MSC k is $\mathsf{inst\_out}_{\mathrm{w}}$-implementable iff the order $<_k^{\mathsf{io}}$ (which is an extension of $<_k^{\mathsf{msc}}$) is cycle-free. Thus checking if an MSC is $\mathsf{inst\_out}_{\mathrm{w}}$-implementable boils down to checking cycle-freeness of this relation. This immediately gives a wide range of efficient implementations for checking class-membership as many algorithms are known in literature for determining whether a given order is cycle-free. For the strong spectrum characterisations are given as well.

Note that the MSCs that distinguish between the different models are surprisingly simple. This indicates that the differences between the classes will appear not only in theory, but also in practice. Besides that, for these distinguishing MSCs, it is not easy to indicate at a glance to which class they do or do not belong. This also supports our view that mechanical support for determining whether a given MSC belongs to a given class is necessary.

Acknowledgements

We would like to thank Thijs Cobben, Loe Feijs, Herman Geuvers and Bart Knaack for their valuable input.

References

[1] M.M. Abdalla, F. Khendek, G. Butler, New results on deriving SDL specifications from MSCs, in: R. Dsoulli, G. von Bochmann, Y. Lahav (Eds.), SDL'99: The Next Millennium, Proc. 9th SDL Forum, Elsevier, Montreal, Canada, 1999.

[2] R. Alur, G.J. Holzmann, D. Peled, An analyzer for message sequence charts, Software Concepts Tools 17 (2) (1996) 70–77.

[3] F. Belina, D. Hogrefe, A. Sarma, SDL—with Applications from Protocol Specification, The BCS Practitioners Series, Prentice-Hall International, London/Englewood Cliffs, 1991.

[4] H. Ben-Abdallah, S. Leue, Syntactic detection of process divergence and non-local choice in message sequence charts, in: E. Brinksma (Ed.), Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, Vol. 1217, Springer, Berlin, 1997, pp. 259–274.

[5] J. Bergstra, J. Klop, Process algebra for synchronous communication, Inform. and Control 60 (1/3) (1984) 109–137.
[6] B. Charron-Bost, F. Mattern, G. Tel, Synchronous, asynchronous and causally ordered communication, Distributed Comput. 9 (4) (1996) 173–191.
[7] W. Damm, D. Harel, LSC's: breathing life into message sequence charts, Formal Methods System Des. 19 (1) (2001) 45–80.
[8] L. Feijs, Generating FSMs from interworkings, Distributed Comput. 12 (1) (1999) 31–40.
[9] J. Grabowski, P. Graubmann, E. Rudolph, Towards a Petri net based semantics definition for message sequence charts, in: O. Færgemand, A. Sarma (Eds.), SDL'93—Using Objects, Proc. 6th SDL Forum, North-Holland, Darmstadt, Germany, 1993, pp. 179–190.
[10] L. Hélouët, C. Jard, Conditions for synthesis of communicating automata from HMSC's, in: S. Gnesi, I. Schieferdecker, A. Rennoch (Eds.), 5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems, Proc. FMICS'2000, Berlin, 2000, pp. 203–224.
[11] S. Heymer, A non-interleaving semantics for MSC, in: Y. Lahav, A. Wolisz, J. Fischer, E. Holz (Eds.), Proc. 1st Workshop of the SDL Forum Society on SDL and MSC, Informatik-Berichte, Vol. 104, Humboldt-Universität zu Berlin, Berlin, 1998, pp. 281–290.
[12] ISO, Information Processing Systems—Open System Interconnection—LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, Vol. DIS 8807, ISO/EC, 1988.
[13] ITU-T, Recommendation Z.100: Specification and Description Language (SDL), ITU-T, Geneva, 1994.
[14] ITU-T, Recommendation Z.120 Annex B: Algebraic semantics of Message Sequence Charts, ITU-T, Geneva, 1995.
[15] ITU-T, Recommendation Z.120: Message Sequence Chart (MSC), ITU-T, Geneva, 1996.
[16] J.-P. Katoen, L. Lambert, Pomsets for message sequence charts, in: Y. Lahav, A. Wolisz, J. Fischer, E. Holz (Eds.), Proc. 1st Workshop of the SDL Forum Society on SDL and MSC, Informatik-Berichte, Vol. 104, Humboldt-Universität zu Berlin, Berlin, 1998, pp. 291–300.
[17] F. Khendek, G. Robert, G. Butler, P. Grogono, Implementability of message sequence charts, in: Y. Lahav, A. Wolisz, J. Fischer, E. Holz (Eds.), Proc. 1st Workshop of the SDL Forum Society on SDL and MSC, No. 104 in Informatik-Berichte, Humboldt-Universität zu Berlin, Berlin, Germany, 1998, pp. 171–179.
[18] I. Krüger, R. Grosu, P. Scholz, M. Broy, From MSCs to statecharts, in: F. Rammig (Ed.), Distributed and Parallel Embedded Systems, Kluwer Academic Publishers, Boston, 1999, pp. 61–71.
[19] P.B. Ladkin, S. Leue, Interpreting message flow graphs, Formal Aspects Comput. 7 (5) (1995) 473–509.
[20] S. Leue, L. Mehrmann, M. Rezai, Synthesizing software architecture descriptions from Message Sequence Chart specifications, in: Proc. 13th IEEE International Conference on Automated Software Engineering, IEEE Computer Society Press, Silver Spring, MD, 1998, pp. 192–195.
[21] N. Mansurov, D. Zhukov, Automatic synthesis of SDL models in use case, in: R. Dsoulli, G. von Bochmann, Y. Lahav (Eds.), SDL'99: The Next Millennium, Proc. 9th SDL Forum, Elsevier, Montreal, Canada, 1999.
[22] S. Mauw, M. van Wijk, T. Winter, A formal semantics of synchronous Interworkings, in: O. Færgemand, A. Sarma (Eds.), SDL'93—Using Objects, Proc. 6th SDL Forum, Darmstadt, Germany, North-Holland, Amsterdam, 1993, pp. 167–178.
[23] S. Mauw, M. Reniers, An algebraic semantics of basic message sequence charts, Comput. J. 37 (4) (1994) 269–277.
[24] S. Mauw, M. Reniers, A process algebra for interworkings, in: J. Bergstra, A. Ponse, S. Smolka (Eds.), Handbook of Process Algebra, Elsevier, Amsterdam, 2000, pp. 1269–1327 (Chapter 19).
[25] M. Reniers, Message sequence chart: syntax and semantics, Ph.D. Thesis, Eindhoven University of Technology, June 1999.
[26] G. Robert, F. Khendek, P. Grogono, Deriving an SDL specification with a given architecture from a set of MSCs, in: A. Cavalli, A. Sarma (Eds.), SDL'97: Time for Testing—SDL, MSC and Trends, Elsevier, Evry, France, 1997, pp. 197–212.
[27] E. Rudolph, P. Graubmann, J. Grabowski, Tutorial on message sequence charts, (Special issue on SDL and MSC, Ø. Haugen (guest editor)) Comput. Networks ISDN Systems 28(12) (1996) 1629–1641.

[28] S. Somé, R. Dssouli, J. Vaucher, From scenarios to timed automata: Building specifications from user requirements, in: Proc. 2nd Asia Pacific Software Engineering Conference, IEEE Computer Society Press, Brisbane, Australia, 1995, pp. 48–57.
[29] S. Somé, R. Dssouli, Using a logical approach for specification generation from message sequence charts, Publication Départementale, 1064, Département IRO, Université de Montréal, April 1997.